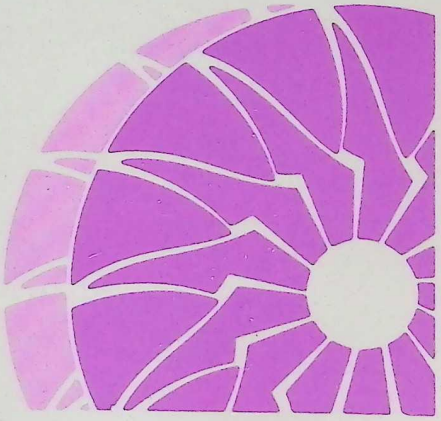
TURBO PASCAL

Volume 1
Screen Management

Version 5.22
S/N: 410771

Turbo Professional®

A library of over 600 essential procedures and functions, optimized for non-object-oriented programming in Turbo Pascal. Includes ■ Pop-up resident routines ■ Windows ■ Menus ■ Interrupt service routines ■ Data entry routines ■ Powerful and useful demo programs and more.



TURBO Power

Turbo Professional

User's Manual

Volume 1
Screen Management

Copyright (c) 1988, 1993, TurboPower Software
All Rights Reserved

Second Edition July 1993

Order line (U.S. and Canada): 800-333-4160
Elsewhere: 719-260-9136

TurboPower Software
P.O. Box 49009
Colorado Springs, CO 80949-9009

Technical Support: 719-260-6641
Fax: 719-260-7151
9 a.m. to 5 p.m. Mountain Time, Monday-Friday

CompuServe 76004,2611
PCVENB Section 6

Turbo Professional is copyright (c) 1987, 1988, 1993 by TurboPower Software, all rights reserved. Portions are copyright (c) 1985, 1986 by Sunny Hill Software, and are used under license to TurboPower Software. All TurboPower Software products are trademarks or registered trademarks of TurboPower Software. Other brand and product names are trademarks or registered trademarks of their respective holders.

Table of Contents

Volume 1: Screen Management

1. Introduction 1
- A. System Requirements 3
- B. Files Supplied 4
- C. Getting Started 6
- D. Purchase Agreement 17

2. Screen Handling 19
- A. Overview 19
- B. TPCRT: Enhanced CRT Unit 21
- C. TPWINDOW: Popup Windows 83
- D. TPMENU: Pulldown Menus 114
- E. TPEDIT: Line Editor 153
- F. TPHELP: Popup Help 168
- G. TPMOUSE: Mouse Routines 199
- H. TPPICK: Pick Lists 231
- I. TPDIR: Popup Directory Picker 252
- J. TPSCREEN: Virtual Screens 260

3. Data Entry 285
- A. Overview 285
- B. TPENTRY: Data Entry Screens 287
- C. TPDATE: Date and Time Variables 378
- D. TPMEMO: Memo Field Editor 402

Identifiers

Index

Volume 2: TSRs and More

4. String Handling 421
- A. Overview 421
- B. TPSTRING: String Manipulation 423
- C. TPASCIIZ: Long ASCIIZ Strings 465
- D. TPSTRDEV: Formatted Strings 484

5. DOS/BIOS Calls 487
- A. Overview 487
- B. TPDOS: Assorted DOS/BIOS Calls 489
- C. TPENHKBD: Enhanced Keyboard Interface 518

6. Interrupts and TSRs 525
- A. Overview 525
- B. TPINT: ISR Management 528
- C. TPTSR: TSR Management 542
- D. TP8087: 8087 Support for TSRs 566
- E. TPINT24: Critical Error Handler 575

7. Sorting 579
- A. Overview 579
- B. TPSORT: In-memory Sort Routine 582

8. Memory Allocation 587
- A. Overview 587
- B. TPEMS: EMS Memory Management 590
- C. TPEXTMEM: Extended Memory Management 602

9. Macros 609
- A. Overview 609
- B. TPMACRO: Keyboard Macros 611
- C. TPMACED: Macro Editor 626

10. Large Arrays 629
- A. Overview 629
- B. TPRARRAY: Arrays in Normal RAM 632
- C. TPEARRAY: Arrays in EMS Memory 644
- D. TPVARRAY: Arrays in Virtual Memory 653

11. BCD Arithmetic **665**
A. Overview 665
B. TPBCD: High-precision Reals 667

12. Miscellaneous **687**
A. Overview 687
B. TPERRHAN: Runtime Error Recovery 688
C. TPINLINE: Assorted Inline Macros 692

13. Demonstration Programs **705**
A. SMACS: Macro Processor 706
B. PCALC: Programmer's Calculator 710
C. PTIME: Programmer's Time Management Utility 715
D. PTRPT: Report Generator for PTIME 718
E. PREF: Programmer's Quick Reference Chart 721
F. MAKEMENU: Menu Maker for TPMENU 724
G. BSORT: File Sorting Utility 735
H. TFIND: Text Find Utility 736
I. REP: Command Repeater 738
J. DIFF: Difference Finder 741
K. VBROWSE: Virtual File Browser 742
L. ENTRY: Data Entry Example 745
M. Other Programs 749

14. Appendices **751**
A. Unit Dependencies 751
B. Keyboard Handling in Turbo Professional 753
C. Using Overlays with Turbo Professional 763

Identifiers

Index

1. Introduction

When we got our first glimpse of Turbo Pascal 4.0, we quickly came to two important conclusions. One, Borland had greatly expanded the potential market for commercial libraries of routines by providing a separate compilation facility. Two, the numerous differences between the Turbo Pascal 4 and Turbo Pascal 3 were going to spell trouble for programmers who had spent months developing and accumulating routines that were optimized for the previous version of the compiler. The best thing we could do for our customers, we thought, would be to develop a powerful library of routines that would allow veteran Turbo Pascal programmers to make a smooth and swift conversion to the new compiler.

The product we developed was Turbo Professional 4.0, the most comprehensive library of Turbo Pascal routines assembled at that time, and it was an instant success--in America and around the world. We've received hundreds of calls and letters from satisfied customers, most of them full of nothing but praise for the package, and we gratefully accept the compliments.

But--there's always a "but"--those same letters almost invariably clamored for even more features. Data entry screens, mouse support, a help facility, exploding windows--the list goes on and on. Some of the requests were filled immediately, but in most cases we had to put people off: "Thank you, we'll try to implement that in a future version."

Then we saw Turbo Pascal 5.0. What we perceived this time was not a need for a new library but rather a golden opportunity to enhance the one we had. Turbo Professional 5.0 was the result, and it contained numerous improvements over the 4.0 version: 600 routines (formerly 400), 31 units (formerly 25), 750 pages of documentation in two volumes (formerly 450 pages in a single volume).

With the release of Turbo Pascal 5.5, TurboPower moved its focus to the new object-oriented programming paradigm, and developed Object Professional which was, and probably still is, the most comprehensive library for writing text-mode DOS applications. Even so, many users of Turbo Professional stuck with it, since it generates smaller applications and doesn't require a switch to the object-oriented programming style.

Over time, more capabilities were added to Turbo Professional, and the READ.ME file and auxiliary DOC files were added to describe these new units and features. As this second edition of the Turbo Professional manual is completed, all of the product's capabilities are once again described in the manual.

In order to get a fast start with the package, we recommend reading Section C of Chapter 1, "Getting Started," which describes how to install Turbo Professional and also provides general information concerning the use of the library.

Before you go, however, be sure to at least glance at the table of contents on the preceding pages to get an idea of how the manual is arranged. Apart from the introductory sections (Chapter 1), Volume 1 deals almost exclusively with routines related to screen handling (Chapters 2 and 3). At the end of the volume you'll find two indexes: a general index concerning key topics, and an index of all the identifiers in the library. Volume 2 begins with a second copy of the table of contents, for your convenience, followed by a series of chapters covering the remaining units in the library (Chapters 4-13). Chapter 14 contains appendices, detailed discussions of broad topics of general interest. The volume ends with copies of the same two indexes found in Volume 1, again for your convenience.

You should also be sure to read the Purchase Agreement in Chapter 1, Section D, which describes your rights as an owner of Turbo Professional, as well as the nature and extent of the warranty that comes with it.

A. System Requirements

To use this package, you must have the following:

1. An IBM PC, XT, AT (or close compatible), or PS/2 running DOS 2.0 or later.

2. Turbo Pascal version 4.0, 5.0, 5.5, 6.0 or 7.0; or Borland Pascal 7.0. You can use Borland Pascal 7.0 to generate DOS real mode or DOS protected mode applications using Turbo Professional.

3. 256K of available memory. (Note, however, that the Turbo Pascal compiler itself may have slightly higher memory requirements.)

4. Two floppy disk drives. A hard disk is desirable, however.

5. An 80-column monitor.

Optional:

6. Borland's Turbo Assembler (TASM) 1.0 or higher, or Microsoft's Macro Assembler (MASM) 4.0 or higher. Needed only if you wish to modify any of the assembly language routines.

B. Files Supplied

Turbo Professional is distributed on dual media: 5.25" 360KB diskettes and 3.5" 720KB diskettes. The number of diskettes of each media type is subject to change. The diskette labeled DEMO contains a series of programs that demonstrate the use of the Turbo Professional library; the diskette labeled HELP/BONUS contains the online reference guide and a "BONUS" subdirectory with various units and programs that have been placed in the public domain; the remaining diskettes are labeled SRC1 through SRCn and contain the source code for the units in the library. Turbo Professional can be installed directly from either set of diskettes or you can copy the complete set of files to your hard disk and run the installation from there.

We'd like to draw your attention to several files in particular:

<u>DEMO Disk:</u>

READ.ME
Read this file before proceeding. It contains the latest updates to this manual.

INSTALL.EXE
Installation program for Turbo Professional.

TPRO.MAK
This file is to be used in conjunction with Borland's MAKE utility (provided with Turbo Pascal). It provides instructions telling MAKE how to rebuild all the Turbo Professional units and demo programs.

PRINTDOC.EXE
A small utility for paginating and printing text files, such as the READ.ME file. Type PRINTDOC with no parameters to get a list of command line options.

LHARC.EXE
A freeware compression/decompression utility used to compress and decompress files with extension LZH. Type LHARC to get a summary of its options.

FASTUPD.nnn
Describes changes in this version of Turbo Professional compared to the most recent previous one. This file is of particular use to Fast Update Plan subscribers. The characters 'nnn' specity the current version number.

HELP/BONUS Disk:

GUIDE.DOC

Documentation for the online reference guide.

BLDGUIDE.EXE

A self-extracting archive that contains the Turbo Professional help text, a precompiled copy of the help compiler (MAKEHELP.EXE), and a TSR help engine (GUIDE.EXE). GUIDE.DOC contains more information.

BONUS

This subdirectory contains various programs and units that have been released into the public domain. Several of them were written by TurboPower Software, others are included with the permission of the author. See the READ.ME! file in this subdirectory for further information. Neither of the installation methods described later (Make File Installation or INSTALL.EXE Installation) installs the files from this subdirectory.

C. Getting Started

Before going any farther, we should probably describe briefly some of the assumptions that were made during the writing of this manual. First and foremost, we assume not only that you are familiar with the Turbo Pascal compiler and how to use it, but also with some of the more fundamental features that Borland added in version 4.0. Most notably, we assume that you are familiar with

- the concept of units (files with the .TPU extension)
- the meaning of the USES statement ("USES CRT,DOS;")
- the distinction between the NEAR and FAR models ({$F+} vs. {$F-})
- the distinction between the interface and implementation sections of a unit
- the use of the new "@" symbol ("MyPointer := @MyVariable;")
- the distinctions between code, data, stack and heap

More generally, if you don't have a basic understanding of units, memory segments, and pointers, you may find some parts of this manual more opaque than they really are. If necessary, please consult the Turbo Pascal manual for the necessary background information. Beyond that, we assume that you have at least a passing familiarity with

- exit procedures ("ExitProc := @MyExitProc;")
- unit initialization blocks (the "begin..end." block at the end of a unit)
- interrupt procedures
- inline macros (you don't have to be able to write them yourself)
- the concept of interrupts and interrupt vectors
- the concept of programs that terminate and stay resident (also known as TSR's or memory resident programs)
- conditional compilation directives ({$IFDEF}, {$ENDIF}, etc.)

This doesn't mean that Turbo Professional is intended strictly for advanced programmers. Quite the contrary, it is consciously designed to be usable by both hobbyists and professional developers. Wherever possible, you are insulated from the gory details of how things actually work. But to some extent we have to assume that we are all speaking the same basic language, if only to avoid insulting your intelligence.

If you need or want further background on the inner workings of PC's, MS-DOS, and/or the IBM BIOS, we recommend the following books:

1. Peter Norton, *Programmer's Guide to the IBM PC* (Microsoft Press). A good general reference book covering all three topics.

2. Ray Duncan, *Advanced MS-DOS* (Microsoft Press). Long the best reference book on DOS and DOS function calls.
3. Terry R. Dettmann, *DOS Programmer's Reference* (Que). A little-known gold mine. Besides the basic stuff covered by Duncan and Norton, this one discusses undocumented DOS calls and other topics such as mouse programming. Highly recommended.
4. Robert Jourdain, *Programmer's Problem Solver for the IBM PC, XT, and AT* (Brady Books). Not a reference book per se, but contains much useful background information.
5. Richard Wilton, *Programmer's Guide to PC and PS/2 Video Systems* (Microsoft Press). Gives very detailed information on the various display adapters.

On to the good stuff...

The Turbo Professional library consists of hundreds of routines in dozens of units, most of which fall into one of several broad categories:

Screen Handling
TPCRT is a plug-in replacement for the standard CRT unit, offering several extra capabilities and a host of enhancements. TPWINDOW contains routines for managing popup windows and virtual screens. TPMENU allows you to create sophisticated menu systems quickly and easily, especially when you use the MAKEMENU utility. TPEDIT provides a full-featured line editor to handle keyboard input. TPHELP allows you to create context-sensitive help systems with a minimum of fuss. TPMOUSE simplifies the task of adding mouse support to your programs. TPPICK displays "pick lists," popup windows containing lists of items from which the user may choose. TPDIR combines with TPPICK to create popup lists of filenames. And TPSCREEN offers an alternative method for working with virtual screens (this unit is included primarily to maintain compatibility with earlier versions of Turbo Professional).

Data Entry
TPENTRY provides a comprehensive full-screen data entry facility. TPDATE provides routines for working with date and time variables. TPMEMO provides a free-form text and memo editor.

String Handling
TPSTRING contains a comprehensive set of basic routines for manipulating standard Turbo strings. TPASCIIZ is a similar unit that allows you to work easily with strings larger than 255 characters. TPSTRDEV lets you use standard Write statements to create formatted strings.

DOS/BIOS Interface
TPDOS gives you easy access to a host of DOS and BIOS services not readily available using the standard DOS unit. TPENHKBD simplifies the task of writing programs that support the enhanced keyboard.

Interrupts and TSRs
TPINT manages your interrupt service routines (ISR's) for you. TPTSR contains everything you need to write a commercial quality TSR with remarkably little effort. TP8087 contains low-level routines needed when writing TSRs that use the 80x87 math chip or Turbo Pascal's floating point emulator, as well as routines that allow you to recover from otherwise fatal arithmetic errors. TPINT24 provides a DOS critical error handler, so your users won't have to Abort, Retry, or Ignore.

Sorting
TPSORT has a fast in-memory sort that works with any data type.

Memory Allocation
TPEMS gives you easy access to expanded memory (EMS), while TPEXTMEM unlocks the door to the protected mode on '286 and '386 machines, letting you use extended memory for data storage.

Macros
TPMACRO lets you easily add a keyboard macro facility to your programs, and TPMACED contains a sophisticated macro editor that goes with it.

Large Arrays
Three units let you break through the 64K barrier on arrays. TPRARRAY uses normal RAM for swift access. TPEARRAY uses EMS memory to allow arrays of up to 32 megabytes. And TPVARRAY stores array elements on disk ("virtual memory"), to permit arrays also as large as 32 megabytes.

BCD Arithmetic
TPBCD provides a comprehensive set of routines for doing high-precision arithmetic using BCD (Binary Coded Decimal) reals.

Miscellaneous Routines
TPERRHAN allows you to recover completely from almost any runtime error. TPINLINE contains an assortment of useful little inline macros for working with pointers, bit flags, and jump tables (among other things).

Now that you have some idea of what's in the library, let's start putting it to work. We'll begin by showing you how to transfer the files from the Turbo Professional distribution diskettes onto your working drive.

TPU files are not distributed with the library. In order to use any Turbo Professional units, you need to compile the appropriate source files. We've provided two different methods for you to do this.

The first method, the easier to use, involves an interactive program called INSTALL.EXE, which you'll find on the DEMO diskette. With this program, you can select any number of the Turbo Professional units, and INSTALL will automatically copy and dearchive them in a location of your choice. INSTALL is especially useful if you don't want to install all of the units, or if you're uncomfortable with Unix-style make files. We'll describe how to use INSTALL in a moment.

The second method, more powerful but less friendly, counts on Borland's MAKE.EXE utility and the supplied make file TPRO.MAK. The make file contains a complete set of dependencies for all of the Turbo Professional files. This method is more desirable if you want to install all of the Turbo Professional tools on a hard disk with at least 3 megabytes of free space. When required, it reassembles OBJ files in addition to running the compiler.

Both installation methods assume you have at least one floppy and a hard disk. If you don't have a hard disk, or if hard disk space is scarce, you need to use INSTALL and carefully select the minimum number of files that will install into the available disk space.

Compiler Support

The Turbo Professional units compensate internally for different compiler versions (see the list of supported compilers in section 1.A) and target environments by the use of conditionally defined code. Conditional logic and defines in the TPDEFINE.INC file are used to get the appropriate behavior. If you have upgraded to this version of Turbo Professional from an earlier version, be certain that you are using the most recent version of TPDEFINE.INC, or you will run into compilation problems.

Protected Mode Operation

When using Borland Pascal 7.0, Turbo Professional can be compiled for real mode or DOS protected mode targets. To support DOS protected mode operation, a new unit, DPMI.PAS, was added to the library. This unit provides low level access to the DOS Protected Mode Interface routines provided by the extender's DPMI server. The routines in this unit are not documented in this manual, but you are welcome to use them with the assistance of a DPMI specification. For more information, refer to the book "*Extending DOS*" by Ray Duncan et al, or the DPMI spec available from Intel or CompuServe.

Some units of Turbo Professional cannot be used in protected mode. You will receive a compiler error message if you attempt to compile them for a protected mode target. The units are:

TPTSR	(TSR engine)
TPEARRAY	(EMS large arrays)
TPEXTMEM	(Raw extended memory access)
TPEMS	(EMS access)

We trust that the reasons for these restrictions are obvious.

There are a few additional restrictions while using other units in protected mode and these are discussed in the appropriate sections that follow.

Make File Installation Method

We'll describe the make file approach first, since you will probably want access to all of Turbo Professional's files. First, choose or create a subdirectory where you want the Turbo Professional files to reside. This directory will hold every file we provide--Pascal and assembly language source code, OBJ and TPU files, as well as the EXE files of the demo programs and utilities. You'll want this directory on the Turbo compiler's search paths for source, OBJ and TPU files. You may also want to put this directory on your DOS path, in order to run the standalone utilities such as SMACS and MAKEMENU. Alternatively, once you've installed Turbo Professional, you can move the EXE files to another directory that's already on the DOS path.

Remember that you need at least 3 megabytes of free disk space to use the make file installation method. (Once installation is complete, you can erase many of the files, reducing the space requirement to less than 500K if you keep just the TPU files around.) If you have less space, skip ahead to the INSTALL program method.

Continuing with the make file installation, you should copy all files from the root directory of each Turbo Professional diskette to the directory you've chosen for Turbo Professional.

One of the files copied is LHARC.EXE. You'll use this file next to decompress the files, by typing:

LHARC *.LZH

The LZH files contain no duplicate files. If LHARC detects that it is about to overwrite an existing file (because your directory wasn't empty to begin with), it will prompt for confirmation before proceeding. If needed, press <CtrlC> to stop LHARC and return to the DOS prompt.

After dearchiving is complete, you can delete the LZH files to regain some disk space:

DEL *.LZH

Before proceeding, you may need to modify TPRO.MAK a little bit for your system. The make file specifies such information as the path to the Turbo Pascal command line compiler and whether you have an assembler or not. The file itself contains all necessary instructions, just use your favorite text editor. If you're not comfortable making minor edits to a make file, you should use the INSTALL.EXE installation method.

Next, compile everything using Borland's MAKE utility:

MAKE -fTPRO.MAK

Don't try to use Microsoft's MAKE (or any other MAKE utility) with the supplied make file. It is sufficiently different from Borland's that it just won't work.

With the correct settings in TPRO.MAK, MAKE will build the entire library without errors. If you obtain any errors, double check the make file. If this doesn't help, check for a TPC.CFG file. (See the Turbo Pascal manual for a description of TPC.CFG.) If you find a TPC.CFG file, erase it or temporarily rename it while MAKE runs.

If you're short on disk space, you can now erase a number of the files (at least until the next time you need to modify or rebuild one of our units). As an optional step, enter

DEL TP*.PAS
DEL TP*.ASM
DEL TP*.IN?
DEL TP*.OBJ
DEL BCD*.*

This leaves only the all-important TPU (or TPP) files, plus the source code and EXE files for the demonstration programs.

If you haven't already done so, take this opportunity to print and read the READ.ME file, which provides information to update this manual.

INSTALL.EXE Installation Method
INSTALL is an interactive tool we've written to make it very easy for you to

dearchive and copy the Turbo Professional units. With it, you can select only as many Turbo Professional files as you actually need. INSTALL even figures out the unit dependencies for you, so you get every file needed to compile a given program.

Before running INSTALL, create a subdirectory on your hard disk where INSTALL will write the Turbo Professional files. INSTALL will later report how many bytes of space it will need for any combination of files you select--if you find you don't have enough disk space available, you can quit from INSTALL and prepare more before continuing.

Next, insert the DEMO disk in drive A and enter

A:INSTALL

When it starts up, INSTALL notes the current drive and directory, and uses that for the default destination (unless the initial drive is A or B, in which case it assumes a destination of C:\TPRO5).

When INSTALL's main window appears, you'll see a screen something like the following:

```
------------------ Turbo Professional Installation Program ------------------

Destination directory : ____________________________________________________
Source directory      : ____________________________________________________

  Disk space free : _________          Disk space required : _________

        Select                      Go                      Quit

Autoselect dependencies: Y                          Overwrite files: Y
       Assembler source: N    Library defines     Confirm overwrite: Y

------------------ Help appears on this line for each topic ------------------
```

The first two line items show the default directories. If you want the Turbo Professional files to go to a different destination, use the cursor keys to position the highlight bar over the directory to change, press <Enter> to select, type in the desired string, and press <Enter> once more to accept the changes. You can also press <Esc> to undo any changes you made while editing an item. Note that both items require a drive and directory name, not a complete pathname.

The next line in the INSTALL command window shows disk space. Disk space free is the approximate number of bytes currently available on the destination drive. This number will be updated as INSTALL copies and compiles files onto that drive. Disk space required shows the minimum number of bytes required to write the currently

selected Turbo Professional files to the destination drive. Of course, you'll want to assure that free space exceeds required space. INSTALL doesn't enforce this requirement, leaving it to you to select a group of files that fit in the available space.

The fourth row, appearing in a different color on your screen, shows INSTALL's "action buttons". When you position the highlight bar over any of these items and press <Enter>, INSTALL takes further action.

The Select action causes INSTALL to display a second window, the File Selection Window. In this window, you can select the Turbo Professional units and demo programs that you'd like to install. Press the cursor keys to position the highlight bar, the <Ins> key to select a particular unit, the <Del> key to deselect a unit, <Enter> to toggle the selection, or <CtrlEnter> to clear all current selections. Selected units are displayed with an arrow character at the left. You'll note that the Disk space required indicator is updated after each selection.

When you select any particular unit or demo program, INSTALL is actually selecting a whole group of files that are required to compile that unit (Pascal files, include files, OBJ files, and so on). Also, if the Autoselect dependencies option is On (more on this below), INSTALL selects any units on which the current program or unit depends. It's actually entertaining to see the units pulled in by each program. You can experiment by alternately selecting and pressing <CtrlEnter>.

Note that automatic dependency selection works only in one direction, however. When you deselect a unit, the units on which it depends are not deselected with it.

This is a good place to reiterate that you should always select the READ.ME file for copying. This file contains information that didn't make it into the printed manual, and it's often very important stuff, indeed. We can't begin to count the number of embarassed customers who have spent money on long-distance support calls only to learn that the answers to their questions were already in the READ.ME file.

Once you've selected the set of files you want, press <Esc> and the Selection window will close, returning you to the Command window.

The other two action buttons, Go and Quit, do what their names suggest. Go starts the process of dearchiving, copying and compiling. Quit returns you to the DOS prompt. Before you actually proceed to the Go action, read on while we describe the rest of INSTALL's options.

The Command window's bottom two rows display toggles controlling how INSTALL behaves. Position the highlight bar over any of these items, and press <Enter> to toggle the setting.

We've already mentioned Autoselect dependencies. When this is set to Y, the Selection window automatically activates any units on which a chosen unit depends.

When Assembler source is set to Y, INSTALL dearchives and copies assembly language source files to the destination. Otherwise, it copies only the pre-assembled OBJ files. You won't need the assembly source unless you want to study or modify it.

When you select Library defines, a window that allows you to specify settings for the conditional compilation defines is displayed. The same keys that are used in the File Selection Window can be used here to activate or deactivate the desired defines.

When Overwrite files is Y, INSTALL will overwrite any file of the same name in the destination directory, subject to your confirmation if the Confirm overwrite setting is also Y. By setting Overwrite files to N, you can easily update your Turbo Professional directory with additional files from the distribution disks without wasting time on files that already exist.

After you've set these options to your liking, and selected the files you want, you're ready to Go. INSTALL will prompt for you to enter the appropriate Turbo Professional diskettes. You'll be able to watch the progress of installation in the DOS window on the lower portion of the screen. If anything goes wrong, INSTALL will prompt whether you want to continue or quit the current set of actions. When INSTALL finishes successfully, press <Esc> to clear the status message, then select the Quit action to return to DOS, or select another group of files.

INSTALL does not compile the Turbo Professional source code for you. Use the supplied TPRO.MAK make file or the compiler's built-in make to build the units.

Writing a Program that uses Turbo Professional

It's a simple matter, really. To incorporate the library routines in your program, you just declare them in your USES statement, then call them as appropriate. Turbo Pascal's compiler and smart linker take care of the rest. Here, for example, is a simple program that uses routines in TPCRT and TPWINDOW:

```
program VirtualScreensDemo;
uses
  TpCrt, TpWindow;
var
  VS : VScreen;
  Row, Col : Byte;
  Key : Word;
  S : String;
begin
  {create a virtual screen of 100 rows and 255 columns}
  if not MakeVScreen(VS, 100, 255) then Halt(1);
  {fill the virtual screen with assorted characters and colors}
```

```
    S[0] := #255;
    for Col := 1 to 255 do S[Col] := Char(Col);
    ActivateVScreen(VS);
    for Row := 1 to 100 do FastWrite(S, Row, 1, MapColor(Row and $F));
    DeactivateVScreen;
    {make a window and display some instructions}
    HiddenCursor;
    Window(1, 1, ScreenWidth, ScreenHeight-1);
    FastWrite('Cursor keys to scroll, <Esc> to quit', ScreenHeight, 22, $E);
    {copy part of the virtual screen to the physical screen}
    MoveVScreenToWindow(VS, 1, 1);
    {process cursor commands until <Esc> is pressed}
    Row := 1; Col := 1;
    repeat
      Key := ReadKeyWord;
      case Key of
        $4800 : {Up}
         if Row > 1 then begin
           Dec(Row);
           MoveVScreenToWindow(VS, Row, Col);
         end;
        $4B00 : {Left}
         if Col > 1 then begin
           Dec(Col);
           MoveVScreenToWindow(VS, Row, Col);
         end;
        $4D00 : {Right}
         if Col < 255-Pred(ScreenWidth) then begin
           Inc(Col);
           MoveVScreenToWindow(VS, Row, Col);
         end;
        $5000 : {Down}
         if Row < 100-(ScreenHeight-2) then begin
           Inc(Row);
           MoveVScreenToWindow(VS, Row, Col);
         end;
      end;
    until Lo(Key) = 27;
    {clean up the screen}
    NormalCursor;
    Window(1, 1, ScreenWidth, ScreenHeight);
    ClrScr;
  end.
```

If you enter and run this program, you'll see that it demonstrates TPWINDOW's ability to manage virtual screens much larger than the physical screen.

Of course, when you begin to write your own programs that use the library, the trick will be in determining what units and library routines you need. In time, you'll be able to do this without assistance, but when you're getting started you'll need some help.

To find the routines and units you need, you have several resources. The individual reference sections (Chapters 2-12) contain detailed discussions of all the routines in

all the units. Within each reference section, the routines are covered in alphabetical order, and the names of the routines appear in large boldface type at the top outside corner of the page. Alternatively, you can use the index of procedures and functions, which is also organized alphabetically; a copy of this index appears at the end of each of the two volumes. This method is more efficient if you know the name of the routine but can't remember how to use it. Or you can use the TSR help engine, POPHELP to browse or search interactively for any of the routines.

For example, let's say that you wanted to write the VirtualScreensDemo program shown above, and that you needed a quick refresher course on the MakeVScreen routine. If you couldn't remember its name, you could turn to the documentation for TPWINDOW in Chapter 2, Section C and flip through the pages until you found what you wanted. Since the names of routines covered on a given page are prominently displayed at the top of the page, this process is faster and more efficient than you might expect. On the other hand, if you knew the name of the routine but not the unit, you could turn to the procedure index, look up MakeVScreen, and find out exactly where you need to go for more information. The entry for MakeVScreen would also contain cross-references to any closely related routines, such as ActivateVScreen.

If you're still a bit uncertain about what Turbo Professional has to offer, there are a couple of things you might want to do. One is to read the overviews for Chapters 2-12. These give more complete sketches of the individual units, and cover some of the issues involved in using them. You might also want to try compiling and running some of the demonstration programs. (A few of them available in pre-compiled form on the DEMO disk.) The documentation for all the demonstration programs can be found in Chapter 13.

D. Purchase Agreement

This software and accompanying documentation are protected by United States copyright law and also by International Treaty provisions. Any use of this software in violation of copyright law or the terms of this agreement will be prosecuted to the best of our ability.

Turbo Professional is copyright (c) 1987, 1988, 1993 by TurboPower Software. Portions are copyright (c) 1985, 1986 by Sunny Hill Software, and are used under license to TurboPower Software.

TurboPower Software authorizes you to make archival copies of this software for the sole purpose of back-up and protecting your investment from loss. Under no circumstances may you copy this software or documentation for the purposes of distribution to others. Under no conditions may you remove the copyright notices made part of the software or documentation.

You may distribute, without runtime fees or further licenses, your own compiled programs based on any of the source code and precompiled units of Turbo Professional. You may not distribute any of the Turbo Professional source code, precompiled units (TPU files), or unmodified example programs without written permission from TurboPower Software.

Note that the previous restrictions do not prohibit you from distributing your own source code or units that depend on Turbo Professional units. However, others who receive your units need their own copies of Turbo Professional in order to write programs that use your units.

The supplied software may be used by one person on as many computer systems as that person uses. We expect that group programming projects making use of this software will purchase a copy of the software and documentation for each member of the group. Contact TurboPower Software for volume discounts and site licensing agreements.

With respect to the physical diskettes and documentation provided with Turbo Professional, TurboPower Software warrants the same to be free of defects in materials and workmanship for a period of 30 days from the date of receipt. If you notify us of such a defect within the warranty period, TurboPower Software will replace the defective diskette(s) or documentation at no cost to you.

TurboPower Software warrants that the software will function as described in this documentation for a period of 30 days from receipt. If you encounter a bug or deficiency, we will require a problem report detailed enough to allow us to find and

fix the problem. If you properly notify us of such a software problem within the warranty period, TurboPower Software will update the defective software at no cost to you.

TurboPower Software further warrants that the purchaser will remain fully satisfied with the product for a period of 30 days from receipt. If you are dissatisfied for any reason, and TurboPower Software cannot correct the problem, contact the party from whom the software was purchased for a return authorization. If you purchased the product directly from TurboPower Software, we will refund the full purchase price of the software upon receipt of the original program diskette(s) and documentation in undamaged condition. TurboPower Software honors returns from authorized dealers, but cannot offer refunds directly to anyone who did not purchase the product directly from us.

TURBOPOWER SOFTWARE DOES NOT ASSUME ANY LIABILITY FOR THE USE OF TURBO PROFESSIONAL BEYOND THE ORIGINAL PURCHASE PRICE OF THE SOFTWARE. IN NO EVENT WILL TURBOPOWER SOFTWARE BE LIABLE TO YOU FOR ADDITIONAL DAMAGES, INCLUDING ANY LOST PROFITS, LOST SAVINGS OR OTHER INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OF OR INABILITY TO USE THESE PROGRAMS, EVEN IF TURBOPOWER SOFTWARE HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

By using this software, you agree to the terms of this section. If you do not agree, you should immediately return the entire Turbo Professional package for a refund.

TurboPower provides CompuServe and telephone support for Turbo Professional on an as-available basis at no extra charge. We encourage you to use CompuServe for support questions. When you call for technical support, the person who happens to answer the phone will try to answer your call on the spot. When you ask a question on CompuServe, the person most qualified to answer your question will respond. For telephone support call on Monday-Friday 9am to 5pm (Mountain time). CompuServe access information and the technical support telephone number are listed on the title page of this manual.

We believe in supporting our customers and will do our best to make this software handle the job it was designed to do!

2. Screen Handling

A. Overview

TPCRT is a plug-in replacement for Turbo Pascal's standard CRT unit. Although completely compatible with CRT where it counts, TPCRT offers an impressive array of extensions and enhancements. You'll find every imaginable routine for moving the cursor and changing its shape; routines to read characters and strings from the screen; routines to write directly to video memory at top speed; routines to save and restore windows; routines to save screens to disk in a compressed format; routines to switch in and out of the EGA's 43-line mode and the VGA's 50-line mode without clearing the screen; routines to poll the keyboard; the list goes on. TPCRT also allows you to write to any video page on a color adapter (even with WriteLn), and it supports multitaskers such as DesqView and OmniView.

TPWINDOW provides a full-blown popup window manager, as well as routines that implement our preferred approach to virtual screens. (TPSCREEN uses a different approach.) The window manager makes it easy to stack windows one on top of the other. Simple function calls let you define a window, remove a window, redisplay a window, destroy a window, bring a window from the bottom of the stack to the top, move or resize a window, etc. Windows can explode onto the screen (with optional sound effects at each level of explosion) and they can have see-through shadows around them. For a dazzling demonstration of what TPWINDOW can do, try running WINWOW.EXE.

Popup menus are here to stay, it appears, and TPMENU can help you do them right without a lot of effort. In keeping with the menu-making spirit, TPMENU gives you lots of options: Borland-style pull-down menus or Lotus-style (horizontal bar) menus; menus that scroll horizontally or vertically; custom menus that display status information about your program; a host of choices for frame types; shadowed menus; exploding menus. All these options can be a bit overwhelming, we admit. But if you find the routines in TPMENU hard to grasp at first, don't worry. The MAKEMENU utility lets you design your menu systems interactively, then turns them into neatly formatted Pascal source code or stores them in a library file to be read at runtime.

TPEDIT contains a small but powerful WordStar-compatible line editor that you can easily incorporate into your programs. More than a glorified version of ReadLn, the ReadString routine implements most of the basic cursor movement and character-, word- and line-oriented commands available in a full-fledged text editor: cursor left and right, cursor to beginning/end of line, cursor left/right one word,

delete character, delete word, delete to end-of-line, delete from beginning of line, restore line, insert and overwrite modes. ReadString can also scroll the editing field horizontally to allow editing of strings wider than the screen. Special versions of the ReadString routine are also included for editing numeric strings (LongInts, Integers, Words, and Reals), as well as two similar routines that you can use when prompting for individual characters.

TPMOUSE offers a comprehensive set of low-level routines that simplify the task of adding mouse support to your programs. You'll find numerous routines for reading the mouse buttons, reading the position of the mouse cursor, hiding and unhiding the mouse cursor, and so on. Better still, several other units in the package--TPEDIT, TPENTRY, TPHELP, TPMENU, and TPPICK--use the routines in TPMOUSE to provide a ready-made mouse interface, simplifying your task even further.

TPHELP provides routines that allow you to develop sophisticated popup help systems quickly and easily. Among other niceties, TPHELP includes automatic indexing of help topics, a facility for embedding links to related topics into the help text, built-in mouse support, and a routine that scans the screen looking for words that match an entry in the help index.

The TPPICK unit manages "pick lists," which you might prefer to think of as "dynamic menus." Whatever you choose to call them, TPPICK provides a convenient means of displaying lists of related items and allowing the user to choose among them.

TPPICK's companion unit, TPDIR, displays a sorted list of the names of all files that meet a specified set of criteria, then allows the user to choose among them. If you're familiar with the file selection facilities in Borland's products and wished you could provide a similar facility in your own programs, then TPDIR is for you.

The TPSCREEN unit sits on top of TPCRT and adds a number of additional features, notably virtual screens. That means that you can define a region of memory as a "screen," write to it while your program has nothing better to do, then pop it onto the actual screen when you're finished. TPSCREEN also has routines that let you move entire windows horizontally and vertically across the screen, copy from screen to screen, fill a window with a particular character or character, and so on. This unit can do a few things with virtual screens that TPWINDOW cannot, and for that reason you may prefer it.

B. TPCRT: Enhanced CRT Unit

As indicated at the start of the chapter, TPCRT is a much enhanced version of Turbo's CRT unit. If you don't use any of its extensions, your programs will almost always behave just as they would if you used CRT instead. There are a few subtle but important differences, however:

(1) TPCRT supports all video pages on color adapters, not just page 0. If you switch video pages by calling SetVisiblePage, you can continue to use Read, Write, GotoXY, and so on. CRT won't let you do that.

(2) Unlike CRT, TPCRT is very much dependent on certain global variables it defines: CurrentPage, CurrentMode, ScreenWidth, ScreenHeight, CurrentDisplay, and VideoSegment. Whereas CRT is constantly calculating the starting address for video memory, for example, TPCRT assumes that the value in VideoSegment is correct. Making such assumptions saves time, but it means that in general you must use TPCRT's routines to change video modes, change display pages, or switch in/out of 43-line/50-line mode. If you use another routine to perform such a task (a routine in a graphics package, for example), you need to then call either GetCrtMode (reinitializes CurrentMode, ScreenWidth, ScreenHeight, CurrentPage, CurrentDisplay, VideoSegment) or ReinitCrt (also reinitializes CheckSnow and the current window coordinates). Also, if you are writing a TSR that uses TPCRT, you need to call one of these two routines on entry to your popup, usually ReinitCrt.

(3) All keyboard input routines are written with TSRs in mind. If you were to write a TSR that used the regular CRT unit's input routines (ReadKey, Read, ReadLn), you would frequently lock out other TSRs: SideKick, for example, would not be able to pop up on top of your TSR while you were waiting for a key. TPCRT avoids this problem.

(4) TPCRT restores the global variable BufLen, available in Turbo Pascal 3.0 but gone in Turbo Pascal 4.0 and higher. This allows you to restrict the length of strings read with Read or ReadLn statements. For example, if you needed to restrict MyString to 10 characters:

```
BufLen := 10;
ReadLn(MyString);
```

BufLen defaults to 126, and it is reinitialized to that value after each Read or ReadLn.

(5) The initialization code for TPCRT automatically recognizes Compaqs and Zeniths (not *all* models, just some), whose display adapters do not require snow

prevention, and sets the CheckSnow flag accordingly. This allows for dramatically improved screen performance on these machines.

(6) TPCRT supports multitaskers such as DesqView and OmniView.

Obviously, these minor differences don't give you much incentive for using TPCRT rather than CRT. The best reasons for doing so can be found in the numerous extensions that it adds. Most fall into one of several basic categories: cursor manipulation, window manipulation, screen reading and writing, and keyboard polling.

One of the largest categories is cursor manipulation. TPCRT makes changing the shape of the cursor a snap with the NormalCursor, FatCursor, BlockCursor, and HiddenCursor routines. You'll also find alternate versions of GotoXY, WhereX, and WhereY that use absolute screen coordinates rather than relative ones (i.e., relative to the current window). Perhaps the handiest of the new cursor routines, though, are GetCursorState and RestoreCursorState, which make it easy to save and restore the current position and size of the cursor.

Of the other extensions, a few deserve special mention. FastWrite is a blazingly fast string writing routine that gives you complete control over the placement of the string and the colors used. MoveScreen rapidly moves blocks of data to and from video memory, without snow (caused by writing to video memory at a bad time) or flicker (caused by turning the "video gun" on and off). SaveWindow and RestoreWindow, which use MoveScreen, let you save and restore windows almost effortlessly, and FrameWindow makes drawing borders around them just as easy.

Special thanks to Carley Phillips for his contributions to TPCRT.

Compatibility with CRT

Although highly compatible with CRT, TPCRT is not interchangeable with CRT as far as the standard units GRAPH3 and TURBO3 are concerned. Both USE CRT, and conflicts will arise if you try to USE either of them in a program that also USES TPCRT, since both CRT and TPCRT would then be present in the same program. To prevent the two units from fighting for control of the screen, TPCRT contains code that checks for the presence of CRT and aborts the program if it is found. An initial check is done when the program starts, and further checks are done later when certain key procedures are called (in case CRT is initialized after TPCRT).

The best solution to this problem is not to use GRAPH3 or TURBO3. These units are provided by Borland to make it easier to convert programs written with earlier versions of the compiler. Once a program has been converted, you would be well advised to remove all dependencies on these units by using the GRAPH unit instead of GRAPH3, and by eliminating the calls to routines in TURBO3.

If this solution is unacceptable, however, there is an alternative. In TPDEFINE.INC, you will find the following statement:

```
{.$DEFINE UseCrt}
```

If you remove the '.' in front of '$' and recompile TPCRT, it will then be completely compatible with CRT. All calls to routines in CRT will be redirected to the equivalent routines in TPCRT, and all global variables that are declared in both units will be shared in common. That is, the variable in TPCRT will be declared absolute on top of the variable of the same name in CRT: for example, 'TextAttr : Byte absolute Crt.TextAttr'. This workaround has several disadvantages, mainly in terms of wasted code space, but it is an acceptable solution in cases where there are no better alternatives. (Special thanks to Lane Ferris and to the folks at Software Science, Inc., formerly known as The Research Group, for their help in developing this workaround.)

The same issue arises, of course, if you try to use TPCRT together with a non-standard unit that USES CRT. This is less of a problem than with GRAPH3 and TURBO3, since you will typically have the source code for the unit, and can simply replace CRT with TPCRT in the USES statement. But you may encounter difficulties when using pre-compiled units distributed in TPU form either as shareware or as commercial software. (Units released into the public domain typically include source code.)

In cases where you don't know what units USE CRT, you may want to use Borland's GREP utility to search all your source files for instances of "CRT". We recommend using the following command line:

GREP -r- -w+ "CRT" *.pas

The '-w+' option is particularly useful, as it will rule out most instances of "TPCRT".

Important note for owners of early versions of Turbo Professional 4.0: TPCRT is *not* incompatible with the standard GRAPH unit, as we originally reported.

Important note for *all* owners of Turbo Professional 4.0: Although the global variables CurrentWidth and CurrentHeight are still supported, they are no longer documented. We recommend that you now use ScreenWidth and ScreenHeight instead. ScreenWidth holds the current width of the screen, just as CurrentWidth does. ScreenHeight holds the current height of the screen (25, 43, 50), whereas CurrentHeight holds the current height less one (24, 42, 49).

Constants, Variables, and Routines Also Found in Turbo's CRT Unit

In general, the following constants, variables, and routines are functionally identical to those in Turbo's standard CRT unit. Only a few special points are worth noting about them:

1. Although TPCRT's TextMode routine reinitializes some important global variables (the same ones that GetCrtMode initializes), it does not initialize all of them. Therefore, after switching video modes, you should call ReinitCrt to insure that they are all initialized properly. TPCRT's TextMode also differs from the standard version in that it allows you to switch into graphics modes with it. (Note that support for graphics in TPCRT in general is severely limited, however.)
2. DirectVideo has the same range of influence that it does in Turbo's CRT unit. That is, it determines how Write/WriteLn, Read/ReadLn, and GotoXY display text on the screen and move the cursor. It does not, however, affect the behavior of TPCRT's non-standard screen writing routines, such as FastWrite and FlexWrite, which *always* write directly to video memory.
3. ClrScr can fill a window or screen using a character other than a space. For example, to fill the screen with asterisks, you would set TextChar to '*' before calling ClrScr. TextChar is automatically reset to ' ' after the screen has been cleared.

Constants

```
Black = 0;                    {color constants}
Blue = 1;
Green = 2;
Cyan = 3;
Red = 4;
Magenta = 5;
Brown = 6;
LightGray = 7;
DarkGray = 8;
LightBlue = 9;
LightGreen = 10;
LightCyan = 11;
LightRed = 12;
LightMagenta = 13;
Yellow = 14;
White = 15;
Blink = 128;

BW40 = 0;                     {video mode constants}
CO40 = 1;
C40 = CO40;
BW80 = 2;
CO80 = 3;
C80 = CO80;
Mono = 7;
Font8x8 = 256;
```

Variables

```
CheckBreak : Boolean;       {enables Ctrl-Break checking}
CheckEOF : Boolean;         {enables Ctrl-Z checking}
CheckSnow : Boolean;        {set to True to prevent snow on CGA's}
DirectVideo : Boolean;      {write directly to screen?}
LastMode : Word;            {current video mode in low byte, 8x8 flag in high}
SaveInt1B : Pointer;        {previous int $1B handler}
TextAttr : Byte;            {current video attribute}
WindMin : Word;             {Window XLow and YLow: in 0..24, 0..79 format}
WindMax : Word;             {Window XHigh and YHigh: in 0..24, 0..79 format}
```

```
procedure AssignCrt(var F : Text);
  {-Routes standard screen I/O through TPCRT rather than DOS}

procedure ClrEol;
  {-Clears the remainder of the current screen line}

procedure ClrScr;
  {-Clears the current window}

procedure Delay(MS : Word);
  {-Delay for MS milliseconds}

procedure DelLine;
  {-Deletes current screen line}

procedure GoToXY(X, Y : Byte);
  {-Moves cursor to column X, row Y, relative to Window}

procedure HighVideo;
  {-Sets intensity bit in TextAttr}

procedure InsLine;
  {-Inserts a new line at the position of the cursor}

procedure LowVideo;
  {-Clears intensity bit in TextAttr}

function KeyPressed : Boolean;
  {-Returns True if a key has been pressed}

procedure NormVideo;
  {-Sets TextAttr to video attribute used at start of program (no longer
    equivalent to HighVideo!)}

procedure NoSound;
  {-Turns off the sound}

function ReadKey : Char;
  {-Reads a character from the keyboard}

procedure Sound(Hz : Word);
  {-Turns on the sound at the designated frequency}

procedure TextBackground(Color : Byte);
  {-Sets background color for screen writes}

procedure TextColor(Color : Byte);
  {-Sets foreground color for screen writes}

procedure TextMode(Mode : Word);
  {-Sets current video mode}
```

```
function WhereX : Byte;
  {-Returns current X (column) coordinate of cursor, relative to Window}

function WhereY : Byte;
  {-Returns current Y (row) coordinate of cursor, relative to Window}

procedure Window(XLow, YLow, XHigh, YHigh : Byte);
  {-Sets current window coordinates}
```

Types

```
deName = array[1..11] of Char;
```

Internal representation of the name associated with a directory entry in a window library. First 8 characters have the name, last 3 have the extension.

```
DirectoryEntry =
 record
  Status : ShortInt;       {0 = in use, -1 = unused, -2 = deleted}
  Name : deName;           {name and extension in FCB format}
  Index : Word;            {index to this member}
  MemberLength : Word;     {# of 128-byte blocks used by member}
  CRC : Word;              {not implemented}
  CreationDate : Word;     {date entry was created--not implemented}
  LastChangeDate : Word;   {date it was last changed--not implemented}
  CreationTime : Word;     {time entry was created--not implemented}
  LastChangeTime : Word;   {time it was last changed--not implemented}
  PadCount : Byte;         {unused bytes in last block of member}
  Filler : array[28..32] of Byte; {padded to 32 bytes}
 end;
```

Format of a directory entry in a library created with CreateLibrary.

```
DirectoryType = array[0..255] of DirectoryEntry;
DirectoryPtr = ^DirectoryType;
```

Format of the directory at the beginning of a library created with CreateLibrary. The actual number of entries in a directory can vary.

```
DisplayType = (MonoHerc, CGA, MCGA, EGA, VGA, PGC);
```

The six standard types of display adapters recognized by TPCRT. MonoHerc means either a plain monochrome adapter or a Hercules monochrome graphics card. PGC stands for IBM Professional Graphics controller card.

```
FlexAttrs = array[0..3] of Byte;
```

An array of attributes for use by the FlexWrite procedure.

```
FrameCharType = (ULeft, LLeft, URight, LRight, Horiz, Vert);
FrameArray = array[FrameCharType] of char;
```

A type of character used by FrameWindow for drawing frames around windows, and an array of such characters.

```
LibName = string[12];
```

Name of a directory entry in a library (a filename).

```
PackedScreen = array[1..4000] of Byte;
```

Holds the contents of a screen after it has been packed with the PackWindow procedure. The actual size of the array varies.

```
PackedWindow =
 record
  Size : Word;   {size of packed window, including this header}
  TopRow : Byte; {coordinates for top left corner of window}
  TopCol : Byte;
  Rows : Byte;   {height of window}
  Cols : Byte;   {width of window}
  AStart : Word; {index to start of attributes section in Contents}
  CDelta : Word; {number of bytes before first PackRec - chars}
  ADelta : Word; {number of bytes before first PackRec - attrs}
  Contents : PackedScreen; {the contents of the packed screen}
 end;
PackedWindowPtr = ^PackedWindow;
```

The format of a screen after it has been packed using the PackWindow procedure. Contents, which holds the packed screen itself, is an array of Size bytes less the size of the first 8 fields in the record (12 bytes).

```
HercCardType = (HercNone, HercPlain, HercPlus, HercInColor);
```

Used to classify the various types of Hercules graphics cards. HercPlain indicates a standard Hercules Graphics Card or compatible. HercPlus indicates a Hercules Graphics Card Plus (with RamFont capabilities). HercInColor indicates a Hercules InColor Card. HercNone is any card that does not fit into one of the other categories.

```
WindowCoordinates =
 record
  XL, YL, XH, YH : Byte;
 end;
```

A record used to save/restore window coordinates.

Constants

```
BiosScroll : Boolean = True;
```

If False, a special scrolling routine is used that suppresses snow and flicker on color graphics adapters. This setting affects the behavior not only of Write and WriteLn statements, but also ScrollWindowUp and ScrollWindowDown. The BIOS scrolling routine is usually faster, but far less appealing to the eye.

```
CrtError : Word = 0;
```

Much like the DosError variable in the DOS unit, this variable is used to return I/O error codes from any of the routines in TPCRT that perform file I/O (CreateLibrary, OpenLibrary, etc.).

```
CrtNotEnoughMem  = $FFFF;
CrtNilPointer    = $FFFE;
CrtNotValidLib   = $FFFD;
CrtWinNotFound   = $FFFC;
CrtDirectoryFull = $FFFB;
CrtBadWindow     = $FFFA;
```

These constants represent all of the non-standard error codes that might be found in CrtError. If CrtError is not 0 and not in the range CrtBadWindow.. CrtNotEnoughMem, then the value is a standard error code returned by IoResult.

```
DetectMultitasking : Boolean = False;
```

If DetectMultitasking is True, TPCRT will perform the checks necessary to determine if the program is being run under a multitasking system such as DesqView or OmniView (formerly TaskView). If it is, MultitaskingOn will be set to True, and all screen writes (using Write/WriteLn, FastWrite, etc.) will be directed to the virtual screen supplied by the system rather than to the physical screen. Since the default setting for DetectMultitasking is False, you need to add two lines to the start of your program if you want it to be able to run as a background task:

```
DetectMultitasking := True;
ReinitCrt;
```

If the program uses TPSCREEN, call ReinitScreen rather than ReinitCrt. This feature allows most programs written with TPCRT and TPSCREEN (and hence TPWINDOW, TPMENU, TPEDIT, and TPENTRY) to be run as background tasks inside a window (the only exceptions we know of are programs that use alternate video pages). They can also be considered "TopView aware" under the definition used by DesqView and OmniView. Note that TPCRT does not support TopView itself, nor does it support Microsoft Windows or Double DOS.

Important note: Do not set DetectMultitasking to True in a TSR. The function call used to determine if a multitasker is in use can cause problems if it is executed inside a partition that has been designated as "writing directly to screen." This is why DetectMultitasking defaults to False. If you know that you won't be writing TSRs, you can change the default to True and recompile TPCRT.

```
DisplayOverride : ShortInt = -1;
```

Because auto-detection of display adapters is an imperfect art, we now provide a means of overriding it: DisplayOverride. This variable is initialized to -1, which enables auto-detection. If you need to force TPCRT to treat, say, an EGA as a CGA, you can set DisplayOverride to the ordinal value of the appropriate DisplayType--"DisplayOverride := Ord(CGA);"--then call ReinitCrt. The more common use for this variable is to force TPCRT to recognize an adapter as an EGA or VGA in cases where auto-detection has failed to identify it correctly. This should rarely be necessary, however.

```
FrameChars : FrameArray = #213#212#184#190#205#179;
```

Used to store the characters used by FrameWindow for drawing frames around windows. It may be altered either directly or indirectly (using SetFrameChars).

```
MapColors : Boolean = True;
```

If True, the MapColor routine will *map* colors (as is done in TPSCREEN) for improved appearance on monochrome and composite black-and-white displays. MapColor is used by MakeWindow (TPWINDOW) and SubMenu (TPMENU), as well as several routines in TPCRT, such as FastWriteWindow. The setting for MapColors does *not* affect the behavior of any routines in the TPCRT unit other than MapColor--except in those cases that are explicitly documented later in the chapter--nor does it affect the behavior of routines in the TPSCREEN unit.

```
TextChar : Char = ' ';
```

Together with TextAttr, this character is used by ClrScr, ClrEol, ScrollWindowUp, and ScrollWindowDown when clearing or scrolling a window. Like BufLen, it is automatically reset to its default value of ' ' after the screen is cleared or scrolled.

Variables

```
BufLen : Word;
```

Maximum length of a string for Read/ReadLn. The valid range is 1..126, and BufLen is always reset to 126 after each Read operation.

```
CtrlBreakFlag : Boolean;
```

Set to True when <CtrlBreak> has been pressed. For internal use only. This variable is exported only so that TPSCREEN can have access to it.

```
CurrentDisplay : DisplayType;
```

The currently selected display adapter. If two display adapters are installed, CurrentDisplay tells you only the type of the adapter currently in use. See also the entry for EnhancedDisplay.

```
CurrentMode : Byte absolute LastMode;
```

The current video mode.

```
CurrentPage : Byte;
```

The current video page.

```
EnhancedDisplay : DisplayType;
```

This variable is meaningful only if it is set to MCGA, VGA, or EGA. It is useful primarily in detecting situations where two adapters are installed, a plain monochrome adapter and an EGA attached to a color monitor, for example. In such a case EnhancedDisplay would be set to EGA, even though CurrentDisplay was set to MonoHerc. This information is especially needed by memory resident programs which want to support the 43- and 50-line modes (which use the 8x8 font) available on the EGA and VGA, respectively. See SMACS.PAS for an example. If an MCGA, VGA, or EGA is not installed, EnhancedDisplay will be set to MonoHerc.

```
InTextMode : Boolean;
```

Set to False when in graphics mode. (Note that Hercules graphics mode is *not* detected.)

```
MultiTaskingOn : Boolean;
```

This variable is set to True if DetectMultitasking is True and a multitasker (DesqView, TaskView/OmniView) was detected.

```
NormalAttr : Byte;
```

The video attribute for NormVideo.

```
OneMS : Word;
```

This variable contains the loop count used by Delay to generate a 1 ms delay.

```
SaveInt1B : Pointer;
```

The previous interrupt $1B handler (both CRT and TPCRT take over this interrupt in order to trap <CtrlBreak>). This variable exists in TPCRT only if it was compiled with Turbo Pascal 4.0. In Turbo Pascal 5.0 or higher, SaveInt1B is located in the SYSTEM unit.

```
ScreenHeight : Word;
```

The current height of the display (usually 25).

```
ScreenWidth : Word;
```

The current width of the display.

```
VideoSegment : Word;
```

The segment portion of the current starting address for video memory. Normally this will be $B000 if a monochrome adapter is in use, or $B800 if a color adapter is in use, but other values are possible if CurrentPage <> 0.

```
VirtualSegment : Word;
```

This variable is normally equal to VideoSegment, but it is sometimes changed by TPWINDOW to allow FastWrite and other screen writing routines to be used on virtual screens. See the documentation for TPWINDOW.

```
VirtualWidth : Word;
```

This variable is normally equal to ScreenWidth, but it is sometimes changed by TPWINDOW to allow FastWrite and other screen writing routines to be used on virtual screens. See the documentation for TPWINDOW.

```
WhichHerc : HercCardType;
```

Indicates the type of Hercules graphics card that is installed and active, if any. It is initialized when the program begins, and it is reinitialized each time that ReinitCrt is called. WhichHerc is set to HercNone if a Hercules card is not installed or not active.

AddWindowToLibrary / BlockCursor

Declaration

```
procedure AddWindowToLibrary(PWP : PackedWindowPtr; var F : file;
                             DP : DirectoryPtr; WinName : LibName);
```

Purpose

Add a packed window to the specified library.

Comments

This routine adds a packed window (PWP^) created with PackWindow to a library (F) already opened with OpenLibrary or CreateLibrary. DP points to the library's directory. WinName is the name to be assigned to the window. It should be in the same form as a filename: that is, a name of up to 8 characters, and an optional extension of up to 3 characters.

If the library already contains an entry of the same name, the existing entry will be marked as deleted, and the new window will be appended to the end of the library.

On exit, CrtError contains the code for any error that might have occurred. Besides the usual I/O error codes, the only possible value for CrtError is CrtDirectoryFull (indicates that the directory for the library is already full).

Example

```
PWP := PackWindow(1, 1, 80, 25);
DP := OpenLibrary(MyLib, 'MYLIB.LBR');
AddWindowToLibrary(PWP, MyLib, DP, 'MYWIN.1');
```

Pack the entire screen, then save it under the name MYWIN.1 in MYLIB.LBR.

See Also

CreateLibrary
OpenLibrary
PackWindow

Declaration

```
procedure BlockCursor;
```

Purpose

Set scan lines for a block cursor.

Comments

This routine changes the cursor scan lines such that the cursor is as large as possible. On machines with a CGA installed, this means that the cursor will use all 8 scan lines; on machines with a monochrome adapter, EGA, or VGA, it will use all 14 scan lines.

See Also

FatCursor
HiddenCursor
NormalCursor
SetCursorSize

ChangeAttribute / ChangeAttributeWindow

Declaration

```
procedure ChangeAttribute(Number : Word; Row, Col, Attr : Byte);
```

Purpose

Change Number video attributes to Attr starting at Row,Col.

Comments

This routine allows you to alter screen colors without altering the text on the screen. Probably the most typical application for ChangeAttribute is in moving bar menus, since it allows you to highlight (or "unhighlight") a particular region of the screen without knowing what it is that's being highlighted. See the demonstration program SMACS.PAS for an example of this technique.

Number specifies the number of video attributes to be changed. Row and Col indicate the starting screen coordinates for the region to be changed. (Note that these screen coordinates are treated as absolute, not relative to the current window.) And Attr specifies the new video attribute for the region.

Example

```
program Colors;
uses TPCRT;
const Attr : Byte = 0;
begin
  ClrScr;
  WriteLn('Hello multi-colored world!'^M^J'Now watch the colors change...');
  Delay(1000);
  while not KeyPressed do begin
    ChangeAttribute(26, 1, 1, Attr and $7F); {suppress blink bit}
    Delay(50); {make it a little easier on the eyes...}
    Inc(Attr); {change the attribute}
  end;
end.
```

Cycles rapidly through all color combinations until a key is pressed.

Declaration

```
procedure ChangeAttributeWindow(Number : Word; Row, Col, Attr : Byte);
```

Purpose

Change Number video attributes to Attr starting at window Row,Col.

Comments

This routine is nothing but a "shell" around the ChangeAttribute routine. It changes the window-relative screen coordinates specified here to absolute screen coordinates, then calls ChangeAttribute. If the global variable MapColors is True, the video attribute Attr is also mapped prior to calling ChangeAttribute.

Note that no range checking is done by ChangeAttributeWindow. It assumes that the window coordinates passed to it are valid, and that all the attributes being changed are within the current window.

CheckKbd / CloseLibrary

Declaration

```
function CheckKbd(var KeyCode : Word) : Boolean;
```

Purpose

Returns True (and the key code) if a keystroke is waiting.

Comments

CheckKbd is similar to KeyPressed, but if a key is waiting it will also return the key code (in the same form as ReadKeyWord). It thus has the advantage of letting you see what the pending key is, without having to call ReadKey or ReadKeyWord, both of which will remove the keystroke from the BIOS's keystroke buffer.

Example

```
var KeyCode : Word;
...
if CheckKbd(KeyCode) then
  if Lo(KeyCode) = 27 then begin  {check for <Esc>}
    KeyCode := ReadKeyWord;       {'eat' the keystroke}
    Exit;                         {and exit}
  end;
```

Check for a pending keystroke; if <Esc> was pressed, 'eat' the keystroke and exit, otherwise continue.

See Also

ReadKeyWord

Declaration

```
procedure CloseLibrary(var F : file; var DP : DirectoryPtr);
```

Purpose

Close library F and deallocate its directory.

Comments

This routine closes a library file (F) previously opened by OpenLibrary, and deallocates the memory allocated for the file's directory (pointed to by DP). The code for any I/O error that occurs when trying to close the file is returned in CrtError. On exit, DP is nil.

Example

```
DP := OpenLibrary(MyLib, 'MYLIB.LBR');
...
CloseLibrary(MyLib, DP);
```

Open MYLIB.LBR, do something, then close it and deallocate the directory.

See Also

OpenLibrary

Declaration

```
function CreateLibrary(var F : file; Name : string;
                       Entries : Byte) : DirectoryPtr;
```

Purpose

Create a library with the specified # of directory entries and return a pointer to its directory.

Comments

This routine is used to create a library file used to store packed windows created with the PackWindow procedure. Each library consists of a directory and a series of library entries (packed windows). F is the library file to open. Name is the filename to be assigned to it. And Entries indicates the maximum number of packed windows that can be stored in the library. It must be in the range 1..255.

Since one entry in the directory is used to store information about the directory itself, CreateLibrary adds 1 to the specified number of Entries, then rounds the result to the next highest multiple of 4. (File space in the library is allocated in blocks of 128 bytes, and a directory entry is 32 bytes; hence the actual number of directory entries must be a multiple of 4.)

Note that the libraries created with this routine use the common LBR file format. Although this format has some weaknesses, it has one important advantage: there are plenty of readily available programs in the public domain that you can use for inspecting and managing these files.

CreateLibrary returns nil in case of error, otherwise a pointer to the directory for the library that was just created. CrtError has the error code if an error occurred. Besides the usual I/O error codes, the only possible value for CrtError is CrtNotEnoughMem (not enough memory to allocate the directory).

Example

```
DP := CreateLibrary(MyLib, 'MYLIB.LBR', 39);
```

Create a library (MyLib) called MYLIB.LBR, with enough room to store 39 entries. DP will point to the library's directory.

See Also

PackWindow

CloseLibrary

CursorEndLine / CursorStartLine

Declaration

```
function CursorEndLine : Byte;
```

Purpose

Returns the ending scan line of the cursor.

Comments

This routine returns the ending scan line for the cursor, which will normally be either 7 (in a color text mode) or 13 (in a monochrome text mode).

Example

```
NormalCursor; {select normal two-line cursor}
WriteLn('Ending scan line for cursor: ', CursorEndLine);
```

This would display either 7 or 13, depending on the current video mode.

See Also

CursorStartLine CursorTypeSL

Declaration

```
function CursorStartLine : Byte;
```

Purpose

Returns the starting scan line of the cursor.

Comments

This routine returns the starting scan line for the cursor, which will normally be either 6 (in a color text mode) or 12 (in a monochrome text mode). In the EGA's 43-line mode, the starting scan line is typically 5.

Example

```
NormalCursor; {select normal two-line cursor}
WriteLn('Starting scan line for cursor: ', CursorStartLine);
```

This would display 5, 6, or 12, depending on the current mode.

See Also

CursorEndLine CursorTypeSL

Declaration

```
function CursorTypeSL : Word;
```

Purpose

Returns a word containing the starting and ending scan lines of the cursor.

Comments

This routine combines the functions of CursorEndLine and CursorStartLine by returning both values. The starting scan line is returned in the high byte of the word, the ending line in the low byte. It can sometimes be used, in combination with SetCursorSize, as a substitute for GetCursorState and RestoreCursorState.

Example

```
var CursorScanLines, Key : Word;
begin
  CursorScanLines := CursorTypeSL;
  HiddenCursor;
  FastWrite('Error. Press any key to continue...', 25, 1, TextAttr);
  Key := ReadKeyWord;
  SetCursorSize(Hi(CursorScanLines), Lo(CursorScanLines));
end;
```

Save the cursor size, hide the cursor, display an error message at the bottom of the screen, wait for a keystroke, then restore the cursor size.

See Also

CursorEndLine
CursorStartLine
GetCursorState
SetCursorSize

DeleteWindowFromLibrary

Declaration

```
procedure DeleteWindowFromLibrary(var F : file; DP : DirectoryPtr;
                                  WinName : LibName);
```

Purpose

Delete a packed window from the specified library.

Comments

This routine deletes the packed window designated by WinName from the specified library (F). DP points to the directory for the library. The library must have previously been opened by a call to CreateLibrary or OpenLibrary. Note that this routine does not physically delete the window from the library; it simply marks it as deleted in the directory. In order to remove the window permanently, PackLibrary must later be called.

On exit, any error code is returned in CrtError. Besides the usual I/O error codes, the only possible value for CrtError is CrtWinNotFound (WinName was not found in the library's directory).

Example

```
DP := OpenLibrary(MyLib, 'MYLIB.LBR');
DeleteWindowFromLibrary(MyLib, DP, 'MYWIN.1');
CloseLibrary(MyLib, DP);
PackLibrary('MYLIB.LBR');
```

Mark MYWIN.1 as being deleted from MYLIB.LBR, then pack the library to make the deletion permanent.

See Also

CreateLibrary
OpenLibrary
PackLibrary

DispPackedWindow / DispPackedWindowAt

Declaration

```
procedure DispPackedWindow(PWP : PackedWindowPtr);
```

Purpose

Display the packed window pointed to by PWP.

Comments

This routine displays a packed window previously created with PackWindow. Note that it is not necessary to indicate where the window is to be displayed, since that information was saved by PackWindow along with the contents of the window.

Example

```
PWP := PackWindow(1, 1, 80, 25);
...
DispPackedWindow(PWP);
```

Save the entire screen as a packed window, do something, then redisplay the packed window.

See Also

PackWindow

Declaration

```
procedure DispPackedWindowAt(PWP : PackedWindowPtr; Row, Col : Byte);
```

Purpose

Display the packed window pointed to by PWP at Row,Col.

Comments

This routine displays the packed window pointed to by PWP at the specified Row and Col(umn). If necessary, the coordinates specified are adjusted to allow the window to fit on the screen. For example, if the window is 40 columns wide, and Col is 70, Col would be changed to 41 (on an 80-column display) to insure that the window fits.

Example

```
PWP := PackWindow(1, 20, 80, 25);
...
DispPackedWindowAt(1, 1);
```

Save the last six rows of the screen as a packed window, then display them on the top six rows of the screen.

DisposePackedWindow / FastCenter

Declaration

```
procedure DisposePackedWindow(var PWP : PackedWindowPtr);
```

Purpose

Dispose of a packed window, setting PWP to nil on exit.

Comments

This routine disposes of a packed window that was created with PackWindow or read in from a file using ReadPackedWindow or ReadWindowFromLibrary. On exit, PWP is set to nil.

Example

```
PWP := ReadPackedWindow('WINDOW.DAT');
DispPackedWindow(PWP);
DisposePackedWindow(PWP);
```

Read in a packed window from a file, display it, then dispose of it.

See Also

PackedWindow
ReadPackedWindow
ReadWindowFromLibrary

Declaration

```
procedure FastCenter(St : string; Row, Attr : Byte);
```

Purpose

Write St centered on window Row in Attr.

Comments

This routine displays the specified string (St) centered within the current window on the specified Row in the designated video attribute (Attr). Row is assumed to be relative to the current window.

Example

```
Window(2, 2, 79, 24);
FastCenter('Hello world!', 1, $F);
```

This would display 'Hello world!' on row 2 (absolute), centered between columns 2 and 79.

See Also

FastFlush

Declaration

```
procedure FastFill(Number : Word; Ch : Char; Row, Col, Attr : Byte);
```

Purpose

Fill Number video words with Ch/Attr starting at Row,Col.

Comments

Like the FillVideoChar routine in TPSCREEN, this routine will fill a contiguous region of video memory starting at Row,Col with Number instances of the specified character/attribute pair (Ch and Attr). A typical use of the routine might be to fill an entire row of the screen with a line-drawing character like #196.

Example

```
FastFill(ScreenWidth, #196, 13, 1, $F);
```

Fill all of row 13 with character #196, displayed in white on black.

See Also

FillVideoChar (TPSCREEN)

Declaration

```
procedure FastFillWindow(Number : Word; Ch : Char; Row, Col, Attr : Byte);
```

Purpose

Fill Number video words with Ch/Attr starting at window Row,Col.

Comments

This routine acts as a "shell" around the FastFill routine. It changes the window-relative screen coordinates specified here to absolute screen coordinates, then calls FastFill to display the character/attribute pairs.

Note that no range checking is done by FastFillWindow. It assumes that the window coordinates passed to it are valid, and that the characters displayed will fit within the current window.

FastFlush

Declaration

```
procedure FastFlush(St : string; Row, Attr : Byte);
```

Purpose

Write St flush right on window Row in Attr.

Comments

This routine displays the specified string (St) flush right against the edge of the current window on the specified Row in the designated video attribute (Attr). Row is assumed to be relative to the current window.

Example

```
Window(2, 2, 79, 24);
FastFlush('Hello world!', 1, $F);
```

This would display 'Hello world!' on row 2 (absolute), with the end of the string ('!') touching the edge of the window at column 79.

See Also

FastCenter

Declaration

```
procedure FastRead(Number, Row, Col : Byte; var St : string);
```

Purpose

Read Number characters from the screen into St starting at Row,Col.

Comments

Like ReadCharAtCursor, FastRead reads text off the screen, but it does so far more rapidly. It is most useful, perhaps, in popups written with the TPTSR unit, which might want to inspect the contents of the screen being displayed by the program that was interrupted.

Number is the number of characters to read off the screen. Row and Col specify the location to start reading from. (Note that these screen coordinates are treated as absolute, not relative to the current window.) And St is a string variable in which the characters read off the screen are returned.

See SMACS.PAS for an example of a practical application for FastRead.

See Also

FastReadLine (TPSCREEN) ReadCharAtCursor

Declaration

```
procedure FastReadWindow(Number, Row, Col : Byte; var St : string);
```

Purpose

Read Number characters from the screen into St starting at window Row,Col.

Comments

This routine acts as a "shell" around the FastRead routine. It changes the window-relative screen coordinates specified here to absolute screen coordinates, then calls FastRead to read the string off the screen.

Note that no range checking is done by FastReadWindow. It assumes that the window coordinates passed to it are valid, and that the string being read off the screen is within the current window.

Example

```
FastWriteWindow('Hello', 1, 1, 7);
FastReadWindow(5, 1, 1, St);
```

This would read the string 'Hello' into St.

FastText / FastTextWindow

Declaration

```
procedure FastText(St : string; Row, Col : Byte);
```

Purpose

Write St at Row,Col without changing the underlying video attribute.

Comments

This routine is identical to FastWrite except that it takes no attribute parameter. Text is displayed in whatever colors happen to be on the screen at the location where the string is written.

Example

```
TextAttr := $F;
ClrScr;
FastText('This text will be displayed in white on black', 1, 1);
```

Display the specified string in the top left corner of the screen in whatever colors are already present on row 1.

See Also

FastWrite

Declaration

```
procedure FastTextWindow(St : string; Row, Col : Byte);
```

Purpose

Write St at window Row,Col without changing the underlying video attribute.

Comments

This routine acts as a "shell" around the FastText routine. It changes the window-relative screen coordinates specified here to absolute screen coordinates, then calls FastText to display the string.

Note that no range checking is done by FastTextWindow. It assumes that the window coordinates passed to it are valid, and that the string will fit inside the window.

Example

```
Window(7, 5, 73, 20);
FastTextWindow('hello', 1, 1);
```

This would display the word 'hello' at row 5, column 7.

Declaration

```
procedure FastVert(St : string; Row, Col, Attr : Byte);
```

Purpose

Write St vertically at Row,Col in Attr.

Comments

This routine is identical to FastWrite except that it writes text vertically rather than horizontally.

Example

```
FastWrite('Writing horizontally...', 1, 2, TextAttr);
FastVert('Writing vertically...', 2, 1, TextAttr);
```

The first string would be written horizontally across the top of the screen, and the second would be written vertically along the left edge of the screen.

See Also

FastWrite

Declaration

```
procedure FastVertWindow(St : string; Row, Col : Byte);
```

Purpose

Write St vertically at window Row,Col.

Comments

This routine is nothing but a "shell" around the FastVert routine. It changes the window-relative screen coordinates specified here to absolute screen coordinates, then calls FastVert to display the string. If the global variable MapColors is True, the video attribute Attr is also mapped prior to calling FastVert.

Note that no range checking is done by FastVertWindow. It assumes that the window coordinates passed to it are valid, and that the string will fit inside the window.

Example

```
Window(7, 5, 73, 20);
FastVertWindow('hello', 1, 1);
```

This would display the word 'hello' vertically in column 7 starting at row 5.

FastWrite

Declaration

```
procedure FastWrite(St : string; Row, Col, Attr : Byte);
```

Purpose

Write St at Row,Col in Attr (video attribute) without snow or flicker.

Comments

This routine offers a convenient means of displaying text on the screen at top speed without causing snow or flicker. Unlike a standard Write statement, FastWrite in itself gives you complete control over where a string is displayed and in what video attribute, eliminating the need for a cumbersome sequence like

```
GotoXY(x,y);
TextColor(n);
TextBackground(n);
Write(s);
```

The only significant limitation of FastWrite is that it can be used only in text modes.

St is the string to be displayed. Row and Col specify where the string is to be displayed. (Note that these screen coordinates are treated as absolute, not relative to the current window.) And Attr is the video attribute in which it is to be displayed. (See the documentation for PREF in Chapter 13, Section E.)

Example

```
var
  Row, I : Byte;
begin
  ClrScr;
  Write('Press any key for a demonstration of fast screen writing...');
  if ReadKeyWord = 0 then {do nothing};
  while not KeyPressed do
    for Row := 1 to ScreenHeight do
      for I := 0 to 7 do
        FastWrite('FastWrite ', Row, Succ(I*10), Lo(Random(127)));
  if ReadKeyWord = 0 then {do nothing};
  ClrScr;
end;
```

For more practical examples of how to use FastWrite, consult the source code for any of the memory resident demonstration programs, such as PREF.PAS.

See Also

FastWriteClip (TPWINDOW) FastWriteWindow (TPWINDOW)

Declaration

```
procedure FastWriteWindow(St : string; Row, Col, Attr : Byte);
```

Purpose

Write a string using window-relative coordinates.

Comments

This routine is nothing but a "shell" around the FastWrite routine. It changes the window-relative screen coordinates specified here to absolute screen coordinates, then calls FastWrite to display the string. If the global variable MapColors is True, the video attribute Attr is also mapped prior to calling FastWrite.

Note that no range checking is done by FastWriteWindow. It assumes the window coordinates passed to it are valid, and that the string will fit inside the window.

In early versions of Turbo Professional, this routine was located in TPWINDOW.

Example

```
Window(7, 5, 73, 20);
FastWriteWindow('hello', 1, 1, $E);
```

This would display the word 'hello' at row 5, column 7. If MapColors were True, and CurrentMode = Mono, the video attribute would be mapped to $F (white on black); otherwise, the specified attribute (yellow on black) would be used.

See Also

FastWrite
FastWriteClip (TPWINDOW)
MapColor

Declaration

```
procedure FatCursor;
```

Purpose

Set larger cursor based on current video mode.

Comments

This routine changes the cursor scan lines such that the cursor is slightly larger (fatter) than the normal two-line cursor. On machines with a color adapter installed, the cursor will use 3 of the 8 available scan lines; on machines with a monochrome adapter, it will use 4 of the 14 available scan lines; and in the EGA's 43-line or the VGA's 50-line mode it will use 5 of the 8 available scan lines.

See Also

BlockCursor
HiddenCursor
NormalCursor
SetCursorSize

FlexWrite

Declaration

```
procedure FlexWrite(St : String; Row, Col : Byte; var FAttrs : FlexAttrs);
```

Purpose

Write St at Row,Col with flexible color handling.

Comments

This routine is similar to FastWrite, but the last parameter, instead of being a single byte that indicates the video attribute for the string, is an array[0..3] of byte. The purpose of the array is to allow different portions of the string to be displayed in different attributes, and the signal used to indicate a change of attributes is a control character: ^A, ^B, ^C.

The default attribute for the string is FAttrs[0]. To change attributes, you would insert a ^A (for FAttrs[1]), ^B (FAttrs[2]), or ^C (FAttrs[3]). To return to the previous attribute, you would insert the same control character a second time. See the examples below.

Note that each time you switch to a new attribute, 2 bytes of stack space are used up. (It is recovered when you restore the previous attribute, or when the routine ends, whichever comes first). It is up to the programmer to insure that adequate stack space is available, although this is rarely a concern.

See PTIME.PAS and PCALC.PAS for further examples of the use of FlexWrite.

Examples

```
const FAttrs : FlexAttrs = ($7, $F, $E, $70);
FlexWrite('This '^A'is'^A' '^B'a'^B' '^C'test', 1, 1, FAttrs);
```

In this example, each of the four words in the string is displayed in a different attribute. The first word and the spaces between words are all displayed in the default attribute (FAttrs[0]).

```
FlexWrite('This '^A'is also a '^B'test', 1, 1, FAttrs);
```

The first word is displayed in FAttrs[0], the next three in FAttrs[1], and the last word in FAttrs[2].

See Also

FastWrite

FlexWriteWindow

Declaration

```
procedure FlexWriteWindow(St : string; Row, Col : Byte; var FAttrs : FlexAttrs);
```

Purpose

Write a string at window Row,Col with flexible color handling.

Comments

This routine is nothing but a "shell" around the FlexWrite routine. It changes the window-relative screen coordinates specified here to absolute screen coordinates, then calls FlexWrite to display the string.

Note that no range checking is done by FlexWriteWindow. It assumes the window coordinates passed to it are valid, and that the string will fit inside the window.

See Also

FlexWrite

Declaration

```
function Font8x8Selected : Boolean;
```

Purpose

Returns True if EGA or VGA is active and using the 8x8 font.

Comments

This routine returns True if an EGA display adapter is active and in 43-line mode, or a VGA adapter is active and in 50-line mode. Calling Font8x8Selected also has the effect of reinitializing the global variable ScreenHeight.

Note: TPCRT's initialization code automatically senses 43-line/50-line mode and sets the default window coordinates to 1,1,80,43/50 if it is found. If your application depends on having a 25-line display, you should add the following code to the beginning of your program:

```
Window(1,1,80,25); {force a 25-line window}
if Font8x8Selected then
  SelectFont8x8(False);
```

Example

```
if CurrentDisplay >= Ega then
  if Font8x8Selected then
    SelectFont8x8(False)
  else
    SelectFont8x8(True);
```

Toggles an EGA from 25- to 43-line (or a VGA from 25- to 50-line) mode or vice versa.

See Also

SelectFont8x8

FrameWindow / GetCrtMode

Declaration

```
procedure FrameWindow(LeftCol, TopRow, RightCol, BotRow, FAttr, HAttr : Byte;
                      Header : String);
```

Purpose

Draws a frame around a window.

Comments

This routine draws a frame (or box) around a window whose dimensions are defined by LeftCol, TopRow, RightCol, and BotRow. (Note that these screen coordinates are treated as absolute, not relative to the current window.) The characters used to draw frames are stored in the global typed constant FrameChars, which can be altered either directly or indirectly (using SetFrameChars). Header is a string to be displayed at the center of the top row of the window. If no header is desired, you can pass an empty string. FAttr is the video attribute to be used in displaying the frame characters, and HAttr is the video attribute for the Header.

Example

```
FrameWindow(1, 1, ScreenWidth, ScreenHeight, TextAttr, TextAttr,
  ' Header Goes Here ');
```

Draws a frame around the entire screen.

See Also

MakeWindow (TPWINDOW) SetFrameChars

Declaration

```
function GetCrtMode : Byte;
```

Purpose

Get the current video mode.

Comments

GetCrtMode also reinitializes key global variables used by TPCRT: CurrentMode (current video mode), InTextMode (False when in graphics mode), ScreenWidth (current width of the screen), CurrentPage (current display page), and VideoSegment (segment portion of starting address for video memory). It is intended primarily to be used in popup routines as a fast alternative to ReinitCrt, which reinitializes several other global variables and so takes longer to execute.

For a good example of a case where GetCrtMode should be used, see PTIME.PAS. PTIME calls it before displaying its on-screen clock, to make certain that (1) the clock is displayed at the top right hand corner of the screen, (2) the display is not in graphics mode, and (3) the clock is displayed on the current video page of the active display adapter.

See Also

Font8x8Selected ReinitCrt

Declaration

```
procedure GetCursorState(var XY, ScanLines : Word);
```

Purpose

Return the current position and size of the cursor.

Comments

This routine, when used in combination with RestoreCursorState, provides a convenient means of saving and later restoring the current size and position of the cursor. It is useful not only in popup routines written with TPTSR, but also in 'black box' routines (e.g., TPEDIT's ReadString) that need to be able to manipulate the size of the cursor without affecting the calling program's display.

Example

```
var XY, CursorScanLines, Key : Word;
begin
  GetCursorState(XY, CursorScanLines);
  HiddenCursor;
  FastWrite('Error. Press any key to continue...', 25, 1, TextAttr);
  Key := ReadKeyWord;
  RestoreCursorState(XY, CursorScanLines);
end;
```

See any of the memory resident demonstration programs for further examples.

See Also

RestoreCursorState

Declaration

```
procedure GotoXYAbs(X, Y : Byte);
```

Purpose

Moves the cursor to column X, row Y.

Comments

Like GotoXY, GotoXYAbs moves the cursor to a specific spot on the screen, but the coordinates passed are treated as absolute, rather than relative to the current window. All coordinates are considered 1-based (top left corner is 1,1), and no range checking is done (the cursor can be moved outside the current window).

Example

```
Window(3,3,80,25);
GotoXYAbs(3,3);
```

Moves cursor to the top left corner of the window, just as 'GotoXY(1,1)' would.

See Also

WhereXAbs
WhereYAbs
WhereXY

HercGraphicsMode / HercModeTestWorks

Declaration

```
function HercGraphicsMode : Boolean;
```

Purpose

Return True if a Hercules card is in graphics mode.

Comments

This routine returns True if a Hercules card (or close compatible) is installed and in graphics mode. Be forewarned that the test used here is far from perfect. It is guaranteed to work for true Hercules cards, and it works with most of the better Hercules clone cards, but there also many cases where it will fail. The most common cases are EGA/VGA clone cards that provide Hercules emulation. To determine whether or not this routine will be reliable, call HercModeTestWorks. Note that it is not possible to determine which graphics page (0 or 1) is active.

Example

```
if HercPresent then begin
  if HercModeTestWorks and HercGraphicsMode then
    SetHercMode(False, 0);
  ...
end;
```

This checks to see if a Hercules card is installed and in graphics mode. If it is, the display is forced back into text mode.

See Also

HercPresent

SetHercMode

Declaration

```
function HercModeTestWorks : Boolean;
```

Purpose

Return True if HercGraphicsMode will work.

Comments

This routine is intended to be used in conjunction with the HercGraphicsMode routine. Its purpose is simply to determine whether or not the test used by that routine is reliable.

See Also

HercPresent

Declaration

```
function HercPresent : Boolean;
```

Purpose

Return true if a Hercules graphics card is present.

Comments

This routine returns True if a Hercules graphics card is installed in the system. Note, however, that this does not necessarily mean that the current display is a Hercules card: on a dual-display system, the Hercules might be the inactive display.

Example

```
if HercPresent then begin
  WriteLn('This program cannot be used on a Hercules display adapter');
  Halt;
end;
```

Halt the program if a Hercules display adapter is installed.

See Also

HercGraphicsMode HercModeTestWorks

HiddenCursor / KbdFlags

Declaration

```
procedure HiddenCursor;
```

Purpose

Hides the cursor.

Comments

This routine hides the hardware cursor on any display adapter. Note, however, that the cursor can sometimes be mysteriously "unhidden" on systems with monochrome adapters installed when the display is scrolled by the BIOS. This is a quirk of the hardware, and it is generally unavoidable.

See Also

BlockCursor
FatCursor
NormalCursor
SetCursorSize

Declaration

```
function KbdFlags : Byte;
```

Purpose

Returns BIOS's keyboard status flags.

Comments

This routine returns the current value of the BIOS's keyboard status flags, a bit-coded byte located at $40:$17. Although this value can be obtained directly by using an absolute variable, the KbdFlags uses the more reliable method of executing BIOS interrupt $16, function 2. The meaning of the individual bits in the flags byte are shown below.

Bit	Val	Meaning
7	$80	Insert on
6	$40	Caps Lock on
5	$20	Num Lock on
4	$10	Scroll Lock on
3	$08	<Alt> depressed
2	$04	<Ctrl> depressed
1	$02	<LeftShift> depressed
0	$01	<RightShift> depressed

Example

```
var
  KbdFlagsByte : Byte;
begin
  KbdFlagsByte := KbdFlags;
  if (KbdFlagsByte and $10) <> 0 then
    WriteLn('Scroll Lock on');
end;
```

Declaration

```
function MapColor(C : Byte) : Byte;
```

Purpose

Map a video attribute for visibility on mono/b&w displays.

Comments

If the global typed constant MapColors constant is False, this function simply returns the input parameter unchanged. Otherwise, if the current video mode (CurrentMode) indicates a monochrome or composite black-and-white display, MapColor will attempt to map the input attribute to a visually appealing value on a display without color. A perfect mapping is unfortunately impossible, but MapColor does a good job in most cases.

The parameter C specifies a combined foreground and background color (blink and background in the top four bits, foreground in the bottom four bits). The parameter you pass should hold the attribute for a full color display. MapColor will return a value mapped for the monochrome display.

The TPWINDOW routine MakeWindow calls MapColor, as does the SubMenu routine in TPMENU and most of the window-relative writing routines in TPCRT. (FlexWriteWindow is a notable exception.) Your program may also call it wherever video attributes are used.

Example

```
FastWrite('hello', 1, 1, MapColor($1C));
```

On a color system, this will display a red 'hello' on a blue background. If MapColors is True and the current video mode is 0, 2, or 7, the string will be dim white on a black background.

Try running the WINWOW demonstration program in color and black-and-white (or monochrome) modes to get a feel for what MapColor does.

See Also

Declarations

MapPackedWindowColors

Declaration

```
procedure MapPackedWindowColors(PWP : PackedWindowPtr);
```

Purpose

Map the colors in a packed window for improved appearance on mono/B&W displays.

Comments

This routine remaps (using MapColor) all the video attributes in a packed window created with PackWindow. Typically, it is to be used in cases where a window was drawn on a color display, saved to a file, then read back from disk (using ReadPackedWindow or ReadWindowFromLibrary) on another machine.

Note that calling this procedure permanently changes the attributes in the packed window, so if you save the window back to a file after remapping the colors, the original color information will be lost.

Example

```
PWP := ReadPackedWindow('WINDOW.DAT');
MapPackedWindowColors(PWP);
DispPackedWindow(PWP);
```

Read the packed window stored in WINDOW.DAT into memory, remap the colors, then display it.

See Also

MapColor
PackedWindow
ReadPackedWindow
ReadWindowFromLibrary

Declaration

```
procedure MoveScreen(var Source, Dest; Length : Word);
```

Purpose

Move Length words from Source to Dest without snow or flicker.

Comments

MoveScreen is similar to Turbo's standard Move procedure, but it is intended to be used strictly for moving data between normal memory and video memory, or from one region of video memory to another. It also differs from the standard Move routine in that it cannot move single bytes of data, only pairs of bytes (words).

Source is the address to move data from, and Dest the address to move data to. If the segment portion of either address is in the region of memory used by a CGA ($B8xx) and CheckSnow is True, MoveScreen will inhibit snow and flicker; otherwise, it will move the data as rapidly as possible. Length is the number of words to be moved.

Example

```
var
  ScreenBufPtr : Pointer;
  BytesOnScreen : Word;
begin
  BytesOnScreen := (ScreenWidth * ScreenHeight) * 2;
  GetMem(ScreenBufPtr, BytesOnScreen);
  MoveScreen(Mem[VideoSegment:0], ScreenBufPtr^, BytesOnScreen div 2);
  {-- do something --}
  MoveScreen(ScreenBufPtr^, Mem[VideoSegment:0], BytesOnScreen div 2);
  FreeMem(ScreenBufPtr);
end;
```

Saves the screen and later restores it. Note that this particular task can be accomplished more easily by using SaveWindow and RestoreWindow.

See Also

RestoreWindow SaveWindow

NormalCursor / OpenLibrary

Declaration

```
procedure NormalCursor;
```

Purpose

Set normal scan lines for cursor based on current video mode.

Comments

This routine restores the cursor to its normal size--two lines, for all display types.

See Also

BlockCursor
FatCursor
HiddenCursor
SetCursorSize

Declaration

```
function OpenLibrary(var F : file; Name : string) : DirectoryPtr;
```

Purpose

Open the specified library and return a pointer to its directory.

Comments

This routine opens a library previously created by CreateLibrary and, if successful, returns a pointer to its directory. If an error occurs, the function returns nil, and CrtError has the error code. Besides the usual I/O error codes, the only possible values for CrtError are CrtNotValidLib (the file isn't a valid library) or CrtNotEnoughMem (insufficient memory to allocate the directory).

Example

```
PWP := PackWindow(1, 1, 80, 25);
DP := OpenLibrary(MyLib, 'MYLIB.LBR');
AddWindowToLibrary(PWP, MyLib, DP, 'MYWIN.1');
```

Pack the entire screen, then save it under the name MYWIN.1 in MYLIB.LBR.

See Also

CreateLibrary

Declaration

```
procedure PackLibrary(LName : string);
```

Purpose

Pack a library to remove deleted entries.

Comments

This routine packs a library created by CreateLibrary to remove any entries that have been marked as deleted by DeleteWindowFromLibrary. LName is the name of the library to pack, and it must be closed at the time of the call to PackLibrary. If an error occurs, CrtError contains the error code. Besides the usual I/O error codes, the possible values for CrtError are CrtNotValidLib (the file isn't a valid library) or CrtNotEnoughMem (insufficient memory to allocate the directory).

When packing the library, PackLibrary creates a temporary file called '$$WORK$$.LBR', copies all undeleted entries in LName into the temporary file, erases LName, then renames the temporary file.

Example

```
DP := OpenLibrary(MyLib, 'MYLIB.LBR');
DeleteWindowFromLibrary(MyLib, DP, 'MYWIN.1');
CloseLibrary(MyLib, DP);
PackLibrary('MYLIB.LBR');
```

Mark MYWIN.1 as being deleted from MYLIB.LBR, then pack the library to make the deletion permanent.

See Also

CreateLibrary

DeleteWindowFromLibrary

PackWindow

Declaration

```
function PackWindow(XLow, YLow, XHigh, YHigh : Byte) : PackedWindowPtr;
```

Purpose

Return a pointer to a packed window, or nil if not enough memory.

Comments

This routine is somewhat similar to the SaveWindow routine, except that it stores the contents of the designated window in a "packed" format. More specifically, it stores the contents of the window as a pair of arrays--one containing the characters, one containing the attributes--and performs run-length encoding on each of the two arrays. This technique allows any sequence of 6 or more identical bytes in the array to be stored as a 5-byte record. Space savings due to compression will typically range from 30 to 90%.

The dimensions of the window to be saved are specified by XLow, YLow, XHigh, and YHigh. This and other information about the window is saved along with the contents of the window, so when you go to redisplay the window (with DispPackedWindow) you don't need to specify the window coordinates a second time. PackWindow returns nil if there is insufficient memory for the packed window, else a pointer to the packed window. The memory allocated for the window may be recovered using DisposePackedWindow.

TPCRT includes a host of routines for displaying packed windows, saving them to files, reading them back from files, and for working with libraries of packed windows.

Protected mode restriction: Creating a PackedWindow via PackWindow is guaranteed to fragment the heap in protected mode, since it is not practical to resize a suballocator heap block.

Example

```
PWP := PackWindow(1, 1, 80, 25);
...
DispPackedWindow(PWP);
DisposePackedWindow(PWP);
```

Save the current window in packed format, redisplay it later, then dispose of it.

See Also

CreateLibrary
DispPackedWindow
DisposePackedWindow
SaveWindow

Declaration

```
function ReadAttrAtCursor : Byte;
```

Purpose

Returns attribute at the current cursor location on the selected page.

Comments

This routine is similar to ReadCharAtCursor, except that instead of returning the character beneath the cursor it returns the video attribute that the character is displayed in.

Example

```
CharAtCursor := ReadCharAtCursor;
AttrAtCursor := ReadAttrAtCursor;
```

Save the character and attribute beneath the cursor in CharAtCursor and AttrAtCursor, respectively.

See Also

ReadAttribute

ReadCharAtCursor

ReadAttribute / ReadAttributeWindow

Declaration

```
procedure ReadAttribute(Number, Row, Col : Byte; var St : string);
```

Purpose

Read Number attributes from the screen into St starting at Row,Col.

Comments

This routine is similar to the FastRead routine except that instead of reading characters off the screen it reads video attributes. It is also similar to ReadAttrAtCursor, but it is much faster.

Number is the number of video attributes to read off the screen. Row and Col specify the location to start reading from. And St is a string variable in which the attributes read off the screen are returned.

Example

```
ReadAttribute(ScreenWidth, 1, 1, S);
for I := 1 to ScreenWidth do
  Byte(S[I]) := SwapNibble(Byte(S[I])) and $7F;
WriteAttribute(S, 1, 1);
```

Reverses all the foreground and background colors on row 1.

See Also

FastRead
ReadAttrAtCursor
WriteAttribute

Declaration

```
procedure ReadAttributeWindow(Number, Row, Col : Byte; var St : string);
```

Purpose

Read Number attributes from the screen into St starting at window Row,Col.

Comments

This routine is nothing but a "shell" around the ReadAttribute routine. It changes the window-relative screen coordinates specified here to absolute screen coordinates, then calls ReadAttribute.

Note that no range checking is done by ReadAttributeWindow. It assumes that the window coordinates passed to it are valid, and that the attributes being read off the screen are within the current window.

See Also

WriteAttribute

Declaration

```
function ReadCharAtCursor : Char;
```

Purpose

Returns character at the current position of the cursor.

Comments

Like FastRead, ReadCharAtCursor can be used to read data off the screen. Although ReadCharAtCursor reads only a single character at a time, and requires that the cursor be positioned properly, it has the advantage of working equally well in the standard CGA graphics modes (assuming that simple Write and WriteLn statements have been used to display text). FastRead can be used only in text modes.

Example

```
var X, Y : Byte;
    XY : Word;
begin
  XY := WhereXY;
  for Y := 1 to ScreenHeight do begin
    for X := 1 to ScreenWidth do begin
      GotoXY(X, Y); Write(Lst, ReadCharAtCursor);
    end;
    WriteLn(Lst);
  end;
  GotoXYAbs(Lo(XY), Hi(XY));
end;
```

This code does the same thing as the BIOS's print screen routine.

See Also

FastRead

ReadKeyWord

Declaration

```
function ReadKeyWord : Word;
```

Purpose

Waits for keypress, then returns scan code and character code together.

Comments

Unlike ReadKey, this routine returns both the scan code and the character code for the key that was pressed. The low byte of the word returned contains the character code, and the high byte contains the scan code.

ReadKeyWord has two potential advantages over ReadKey. First, it allows you to get all the information you need with a single function call. Second, it allows you to distinguish between keys that share the same character code but different scan codes. For example, on a standard PC or AT keyboard, there are two keys that return '+' (#43): one on the top row of the main cluster of keys, and one on the numeric keypad. The first one has a scan code of 13 ($0D), and the second (sometimes referred to as the 'gray plus key') has a scan code of 78 ($4E). Using ReadKeyWord, you can easily distinguish the two if desired, as is shown in the example below.

Example

```
var
  Key : Word;
  Ch : Char absolute Key;
begin
  Key := ReadKeyWord;
  case Ch of
    '+' : if (Hi(Key) = $4E) then
            WriteLn('Gray plus key pressed')
          else
            WriteLn('White plus key pressed');
  end;
end;
```

For further examples, see SMACS.PAS, which uses ReadKeyWord frequently.

See Also

CheckKbd

Declaration

```
function ReadPackedWindow(FName : string) : PackedWindowPtr;
```

Purpose

Read the packed window stored in FName into memory and return a pointer to it.

Comments

This routine reads into memory a packed window previously saved to disk (as FName) by WritePackedWindow. If successful, it returns a pointer to the packed window. If an error occurs, it returns nil, and CrtError has the error code. Besides the usual I/O error codes, the possible values for CrtError are CrtBadWindow (the file indicated by FName does not contain a valid packed window) and CrtNotEnoughMem (not enough memory to allocate the packed window).

Example

```
PWP := ReadPackedWindow('WINDOW.DAT');
DispPackedWindow(PWP);
DisposePackedWindow(PWP);
```

Read in a packed window from a file, display it, then dispose of it.

See Also

PackWindow

WritePackedWindow

ReadWindowFromLibrary

Declaration

```
function ReadWindowFromLibrary(var F : file; DP : DirectoryPtr;
                               WinName : LibName) : PackedWindowPtr;
```

Purpose

Read a packed window from a library and return a pointer to it.

Comments

This routine loads into memory a packed window that has been stored in a library using AddWindowToLibrary. F is the library file, DP is a pointer to its directory, and WinName is the name of the entry to be read into memory.

If successful, a pointer to the packed window is returned. If an error occurs, it returns nil, and CrtError has the error code. Besides the usual I/O error codes, the only possible values for CrtError are CrtNotEnoughMem (not enough memory to allocate the packed window) and CrtBadWindow (the file indicated by FName does not contain a valid packed window).

Example

```
DP := OpenLibrary(MyLib, 'MYLIB.LBR');
PWP := ReadWindowFromLibrary(MyLib, DP, 'MYWIN.1');
DispPackedWindow(PWP);
DisposePackedWindow(PWP);
CloseLibrary(MyLib, DP);
```

Read in a packed window from a library, display it, then dispose of it and close the library.

See Also

AddWindowToLibrary
CreateLibrary
OpenLibrary
CloseLibrary

Declaration

```
procedure ReinitCrt;
```

Purpose

Reinitializes TPCRT's internal variables.

Comments

ReinitCrt is intended primarily for use in 1) TSR's, 2) programs that use ExecDos or Exec, and 3) programs that use graphics. It completely reinitializes all of the mode-sensitive global variables used by TPCRT: CurrentMode (current video mode), InTextMode (False when in graphics mode), ScreenWidth (current width of the screen), ScreenHeight (current height of the screen), WindMin and WindMax (window dimensions--the entire screen is made the current window), CurrentPage (current video page), CurrentDisplay (the type of the current display adapter), CheckSnow (inhibit snow on color displays), and VideoSegment (segment portion of the starting address for video memory).

In general, there are only three situations in which you should need to call ReinitCrt: on entry to a popup, after executing a DOS call with ExecDos (or Turbo's Exec routine), or after switching into or out of graphics mode.

See Also

ExecDos (TPDOS)

GetCrtMode

Declaration

```
procedure RestoreCursorState(XY, ScanLines : Word);
```

Purpose

Reset the cursor to a position and size saved with GetCursorState.

Comments

This routine, when used in combination with GetCursorState, provides a convenient means of saving and later restoring the current size and position of the cursor. It is particularly useful not only in popup routines written with the TPTSR unit, but also in 'black box' routines (such as TPEDIT's ReadString) that need to be able to manipulate the size of the cursor without having the change affect the calling program's display.

See the entry for GetCursorState for an example.

See Also

GetCursorState

RestoreWindow / RestoreWindowCoordinates

Declaration

```
procedure RestoreWindow(XLow, YLow, XHigh, YHigh : Byte; Deallocate : Boolean;
                        var Covers : Pointer);
```

Purpose

Restore screen contents and deallocate buffer space if requested.

Comments

When used in conjunction with SaveWindow, this routine provides a convenient means of saving and restoring all or part of the screen (a window).

The dimensions of the window to be restored are defined by XLow, YLow, XHigh, and YHigh. (Note that these screen coordinates are treated as absolute, not relative to the current window.) Although the window coordinates will ordinarily be the same as those passed to SaveWindow, they don't have to be. Covers is a pointer to the buffer being used to store the screen data. Deallocate indicates whether the buffer should be deallocated after the window is restored.

See the entry for SaveWindow for an example, as well as for further details.

See Also

EraseWindow (TPWINDOW)
MoveScreen
SaveWindow

Declaration

```
procedure RestoreWindowCoordinates(WC : WindowCoordinates);
```

Purpose

Restore previously saved window coordinates.

Comments

This routine allows you to restore window coordinates previously saved using SaveWindowCoordinates. See the entry for StoreWindowCoordinates for an example.

See Also

RestoreWindowCoordinates (TPSCREEN)
StoreWindowCoordinates

Declaration

```
function SaveWindow(XLow, YLow, XHigh, YHigh : Byte; Allocate : Boolean;
                    var Covers : Pointer) : Boolean;
```

Purpose

Allocate buffer space if requested and save window contents.

Comments

When used in conjunction with RestoreWindow, this routine provides a convenient means of saving and restoring all or part of the screen (a window). The dimensions of the window to be saved are defined by XLow, YLow, XHigh, and YHigh. (Note that these screen coordinates are treated as absolute, not relative to the current window.) Allocate indicates whether or not a buffer should be allocated for the screen data. If Allocate is True, a pointer to the buffer is returned in Covers. Otherwise, Covers must be a pointer to a buffer large enough to store the data. SaveWindow will return False only if Allocate is True and insufficient memory remains on the heap to allow a buffer to be allocated.

Since SaveWindow uses MoveScreen to move data, snow and flicker are inhibited if necessary.

Note: SaveWindow and RestoreWindow are intended to be used only while in text modes, not graphics modes.

Examples

```
var
 BufPtr : Pointer;
begin
 if not SaveWindow(1, 1, ScreenWidth, ScreenHeight, True, BufPtr^) then
   Exit;
 {--do something--}
 RestoreWindow(1, 1, ScreenWidth, ScreenHeight, True, BufPtr);
end;
```

Incorrect! The '^' in 'BufPtr^' is not needed.

```
if not SaveWindow(1, 1, ScreenWidth, ScreenHeight, True, BufPtr) then
  Exit;
```

Correct. This would allocate a buffer and save the entire screen, later restoring it with RestoreWindow.

See Also

MoveScreen

RestoreWindow

ScrollWindowDown

Declaration

```
procedure ScrollWindowDown(XLo, YLo, XHi, YHi, Lines : Byte);
```

Purpose

Scrolls the designated window down the specified number of lines.

Comments

This routine provides an alternative to InsLine in cases where you don't want to move the cursor, or you need to scroll a region of the screen other than the current window.

The window to be scrolled is defined by XLo, YLo, XHi, and YHi. (Note that these screen coordinates are treated as absolute, not relative to the current window.) Lines is the number of lines to be scrolled. If Lines = 0, the entire window is cleared.

Like InsLine, ScrollWindowDown fills the blank lines it creates with spaces, which are displayed in the current video attribute (TextAttr).

Note that, if BiosScroll is false, a special scrolling routine is used that inhibits snow and flicker on color graphics adapters.

Example

```
var
  I : Word;
begin
  ClrScr;
  for I := 1 to ((ScreenHeight-2) * ScreenWidth) do
    Write('*');
  FrameWindow(1, 1, ScreenWidth, ScreenHeight-2, TextAttr, TextAttr,
    ' ScrollWindowDown Demonstration ');
  Delay(2000);
  ScrollWindowDown(2, 2, Pred(ScreenWidth), ScreenHeight-3, 1);
end;
```

Fills all but the last two lines of the screen with asterisks, draws a frame around the region, then scrolls the window inside the frame down one line.

See Also

ScrollWindowUp

Declaration

```
procedure ScrollWindowUp(XLo, YLo, XHi, YHi, Lines : Byte);
```

Purpose

Scrolls the designated window up the specified number of lines.

Comments

This routine provides an alternative to DelLine in cases where you don't want to move the cursor, or you need to scroll a region of the screen other than the current window.

The window to be scrolled is defined by XLo, YLo, XHi, and YHi. (Note that these screen coordinates are treated as absolute, not relative to the current window.) Lines is the number of lines to be scrolled. If Lines = 0, the entire window is cleared.

Like DelLine, ScrollWindowUp fills the blank lines it creates with spaces, which are displayed in the current video attribute (TextAttr).

Note that, if BiosScroll is false, a special scrolling routine is used that inhibits snow and flicker on color graphics adapters.

Example

```
var
  I : Word;
begin
  ClrScr;
  for I := 1 to ((ScreenHeight-2) * ScreenWidth) do
    Write('*');
  FrameWindow(1, 1, ScreenWidth, ScreenHeight-2, TextAttr, TextAttr,
    ' ScrollWindowUp Demonstration ');
  Delay(2000);
  ScrollWindowUp(2, 2, Pred(ScreenWidth), ScreenHeight-3, 1);
end;
```

Fills all but the last two lines of the screen with asterisks, draws a frame around the region, then scrolls the window inside the frame up one line.

See Also

ScrollWindowDown

SelectFont8x8

Declaration

```
procedure SelectFont8x8(On : Boolean);
```

Purpose

Toggle 8x8 font (43-line mode or 50-line mode) on or off.

Comments

This routine allows you to switch in (On = True) and out (On = False) of the EGA's 43-line mode and the VGA's 50-line mode. If an EGA or VGA is not installed, it does nothing. After the switch into 43-line/50-line mode has been made, the global variable ScreenHeight will be set to 43 or 50 rather than its usual value of 25. The current window coordinates, however, will not be changed, since doing so is not always desirable. Changing them to reflect the new screen dimensions is the responsibility of the programmer.

This routine does *not* clear the screen, as does the only method offered by the standard CRT unit:

```
TextMode(CurrentMode+Font8x8);
```

Note: Due to a flaw in the EGA BIOS, a normal two-line cursor (with a starting scan line of 6, and an ending scan line of 7) will be hidden after switching to 43-line mode if CGA cursor emulation is left on. To prevent this, SelectFont8x8 automatically turns emulation off when switching into 43-line mode, and turns it on again when switching back into 25-line mode.

Example

```
if not CurrentDisplay < Ega then
  WriteLn('EGA or VGA required')
else begin
  Write('This is 25-line mode... ');
  Delay(1000);
  SelectFont8x8(True);
  WriteLn('and this is ', ScreenHeight, '-line mode');
  Delay(1000);
  SelectFont8x8(False);
end;
```

See Also

Font8x8Selected

Declaration

```
procedure SetBlink(On : Boolean);
```

Purpose

Enable text mode attribute blinking if On is True.

Comments

SetBlink enables or disables blinking text. If it is off, "bright" colors such as pink ($D) and yellow ($E) may be used as background colors.

To see what effects can be achieved with SetBlink(Off), load the demonstration program PREF, press <Ctrl><RightShift><R> to pop it up, <CtrlPgDn> to get to the end of the table, then <AltB> to turn blinking off. All values greater than 127 will then correspond to colors with a brighter background.

Example

```
TextAttr := $8F;
WriteLn('This produces blinking text');
SetBlink(False);
WriteLn('This doesn''t, even though the attribute is the same');
```

The first statement is displayed in white on a blinking black background. The second is displayed in White on Dark Gray.

Declaration

```
procedure SetCrtBorder(Attr : Byte);
```

Purpose

Set screen border to Attr if card type and mode allow.

Comments

SetCrtBorder allows you to set the color of the screen border to the specified video attribute (Attr) on color adapters (CGA, MCGA, EGA, VGA). It does nothing if the current display is a mono card. Be forewarned that the results are less than impressive on EGA cards.

Example

```
TextColor(White);
TextBackground(Blue);
ClrScr;
SetCrtBorder(Blue shl 4);
```

This sets the screen colors to white on blue, and the screen border to blue.

SetCursorSize / SetFrameChars

Declaration

```
procedure SetCursorSize(StartLine, EndLine : Byte);
```

Purpose

Sets the cursor's starting and ending scan lines.

Comments

Although other procedures in TPCRT allow you to change the size of the cursor, this is the only one that gives you complete control over the starting and ending scan lines. In monochrome text mode (7), the cursor box contains 14 scan lines, numbered from top (start) to bottom (end) as 0-13. In color text modes, the cursor box contains 8 scan lines, numbered from top to bottom as 0-7. Note that, although the EGA's cursor box has 14 scan lines, it normally emulates the CGA (for compatibility), and so can be treated as having only 8 scan lines in text modes.

Example

```
if CurrentMode = 7 then
  EndLine := $0D  {= 13}
else
  EndLine := $07; {= 7}
SetCursorSize(0, EndLine);
```

This would produce a "block cursor" (see the entry for BlockCursor).

See Also

BlockCursor
FatCursor
HiddenCursor
NormalCursor

Declaration

```
procedure SetFrameChars(Vertical, Horizontal, LowerRight, UpperRight, LowerLeft,
                        UpperLeft : Char);
```

Purpose

Sets the frame characters to be used on subsequent FrameWindow calls.

Comments

This routine allows you to change the characters used by FrameWindow to draw a box (or frame) around a window. If desired, you can also change one or more of these characters directly by altering the global variable FrameChars.

Example

```
SetFrameChars(#179, #196, #217, #191, #192, #218);
FrameWindow(1, 1, ScreenWidth, ScreenHeight, TextAttr, TextAttr,
  ' Header Goes Here ');
```

Draws a single-line frame around the entire screen.

See Also

FrameWindow

Declaration

```
procedure SetHercMode(GraphMode : Boolean; GraphPage : Byte);
```

Purpose

Set Hercules card to graphics mode or text mode.

Comments

This routine allows you to switch a Hercules card into graphics mode (GraphMode = True) or text mode (GraphMode = False). If GraphMode is True, GraphPage specifies which graphics page is to be made the active page (0 or 1). The GraphPage parameter is ignored if GraphMode is False.

Example

```
HercGraph := HercGraphicsMode;
if HercGraph then SetHercMode(False, 0);
...
if HercGraph then SetHercMode(True, 0);
```

Switch a Hercules graphics card into text mode, then back into graphics mode.

See Also

HercGraphicsMode HercPresent

Declaration

```
procedure SetVisiblePage(PageNum : Byte);
```

Purpose

Set the current video page.

Comments

This routine allows you to set the current video page on a color adapter (CGA, EGA, MCGA, or VGA). It has no effect on monochrome adapters. In 40-column text modes, there are 8 video pages available (numbered 0-7). In 80-column text modes, there are 4 (numbered 0-3). Note that the current video page is always stored in the global variable CurrentPage.

Example

```
if CurrentMode = 7 then
  WriteLn('Color adapter required')
else begin
  WriteLn('This is video page 0'); Delay(1000);
  SetVisiblePage(1);
  ClrScr;
  WriteLn('This is video page 1'); Delay(1000);
  SetVisiblePage(0);
end;
```

See Also

SetActivePage (TPSCREEN)

StoreWindowCoordinates

Declaration

```
procedure StoreWindowCoordinates(var WC : WindowCoordinates);
```

Purpose

Store the window coordinates for the active window}

Comments

This routine allows you to save the window coordinates for the active window. They can later be restored using RestoreWindowCoordinates.

Example

```
var WC: WindowCoordinates;
begin
  StoreWindowCoordinates(WC);
  Window(1, 1, 80, 10);
  ...
  RestoreWindowCoordinates(WC);
end;
```

See Also

RestoreWindowCoordinates

StoreWindowCoordinates (TPSCREEN)

Declaration

```
procedure StuffKey(W : Word);
```

Purpose

Stuff one key into the keyboard buffer.

Comments

This routine inserts the specified key code directly into the keyboard buffer. If the keyboard buffer is full, it does nothing.

Examples

```
StuffKey($3B00);
```

Inserts the key code for <F1> into the keyboard buffer.

```
StuffKey(Ord('A'));
```

Inserts the key code for <A> into the keyboard buffer. (Most programs ignore the high byte of the key code if the low byte is non-zero.)

Declaration

```
procedure StuffString(S : string);
```

Purpose

Stuff the contents of S into the keyboard buffer.

Comments

This routine inserts the specified string directly into the keyboard buffer. The string must be a maximum of 15 characters long (assuming that the keyboard buffer is empty to begin with). Excess characters are ignored. Along with StuffKey, this routine can be used to implement a simple keyboard macro system, as well as to pass keystrokes to programs being executed with Exec (DOS) or ExecDos (TPDOS).

Example

```
StuffString('Hello world!'^M);
ReadLn(MyString);
```

This would load MyString with 'Hello world!'.

SwitchInColorCard

Declaration

```
procedure SwitchInColorCard(ColorOn : Boolean);
```

Purpose

Activate or deactivate colors on a Hercules InColor card.

Comments

This routine activates or deactivates CGA emulation on a Hercules InColor Card. It should be called only if the global variable WhichHerc is set to HercInColor.

Note that this routine is called automatically by the TPCRT initialization code to activate CGA emulation. Consequently, all you need to do to support the InColor card in most programs is check the value of WhichHerc when selecting the colors for text display. If its value is HercInColor, then you should select the same colors that you would use on a CGA or other color card, even if the current video mode is 7.

SwitchInColorCard needs to be called only if you wish to deactivate CGA emulation, or if you need to reenable it after executing another program (for example).

Example

```
if WhichHerc = HercInColor then
  SwitchInColorCard(False);
{... execute the other program ...}
if WhichHerc = HercInColor then
  SwitchInColorCard(True);
```

Switch back to monochrome colors, execute another program, then reenable CGA emulation.

Declaration

```
function WhereXAbs : Byte;
```

Purpose

Return absolute X coordinate (column) of cursor.

Comments

Like the standard WhereX function, this routine returns the current X coordinate (column) of the cursor, but the number is absolute, rather than relative to the current window. As with WhereX, all X coordinates returned are 1-based (top left corner of screen is 1,1).

Example

```
Window(3,3,80,25);
GotoXY(1,1);
```

In this case, WhereX would return 1, and WhereXAbs would return 3.

See Also

WhereXY

WhereYAbs

Declaration

```
function WhereXY : Word;
```

Purpose

Returns the absolute X (column) and Y (row) coordinates of the cursor.

Comments

Combines the functions of WhereXAbs and WhereYAbs, allowing you to get both values with a single call. The high byte of the result has the current Y coordinate (row), low byte has the current X coordinate (column). As with WhereXAbs and WhereYAbs, all coordinates returned are 1-based (top left corner of screen is 1,1).

Example

```
Window(3,3,80,25);
GotoXY(1,1);
```

In this case, WhereX and WhereY would return 1, WhereXAbs and WhereYAbs would return 3, and WhereXY would return $0303 (771).

See Also

WhereXAbs

WhereYAbs

WhereXYdirect

Declaration

```
procedure WhereXYdirect(var X, Y : Byte);
```

Purpose

Read the current position of the cursor directly from the CRT controller.

Comments

Unlike other TPCRT routines that read the cursor position, this one reads directly from the 6845 CRT controller using Port instructions, bypassing the BIOS. This makes WhereXYdirect extremely hardware dependent, but it also allows you to read the cursor position accurately in cases where other routines fail. The most common application for WhereXYdirect is in TSRs, which must sometimes pop up over applications that do not use the BIOS to position the cursor. The best known example of such a program is Lotus 1-2-3, but there are many others.

Example

```
var
  SaveX, SaveY : Byte;
begin
  {get the cursor position}
  WhereXYdirect(SaveX, SaveY);
  {update the BIOS's cursor data}
  GotoXYabs(SaveX, SaveY);
  ...
  {restore the cursor to its previous position}
  GotoXYabs(SaveX, SaveY);
end;
```

See Also

GotoXYabs

WhereXY

Declaration

```
function WhereYAbs : Byte;
```

Purpose

Returns absolute Y coordinate (row) of the cursor.

Comments

Like the standard WhereY function, this routine returns the current Y coordinate (row) of the cursor, but the number is absolute, rather than relative to the current window. As with WhereY, all Y coordinates returned are 1-based (top left corner of screen is 1,1).

Example

```
Window(3,3,80,25);
GotoXY(1,1);
```

In this case, WhereY would return 1, and WhereYAbs would return 3.

See Also

WhereXAbs WhereXY

Declaration

```
procedure WriteAttribute(St : String; Row, Col : Byte);
```

Purpose

Write string of attributes St at Row,Col without changing characters.

Comments

This routine takes as input a string of video attributes and writes them to the the screen starting at Row,Col. The characters on the screen are not disturbed. WriteAttribute is intended to be used in conjunction with ReadAttribute.

Example

```
ReadAttribute(ScreenWidth, 1, 1, S);
for I := 1 to ScreenWidth do
  Byte(S[I]) := SwapNibble(Byte(S[I])) and $7F;
WriteAttribute(S, 1, 1);
```

Reverses all the foreground and background colors on row 1.

See Also

ReadAttribute WriteAttributeWindow

WriteAttributeWindow / WritePackedWindow

Declaration

```
procedure WriteAttributeWindow(St : String; Row, Col : Byte);
```

Purpose

Write string of attributes St at window Row,Col without changing characters.

Comments

This routine is nothing but a "shell" around the WriteAttribute routine. It changes the window-relative screen coordinates specified here to absolute screen coordinates, then calls WriteAttribute.

Note that no range checking is done by WriteAttributeWindow. It assumes that the window coordinates passed to it are valid, and that the attributes being written to the screen will be within the current window.

See Also

ReadAttributeWindow WriteAttribute

Declaration

```
procedure WritePackedWindow(PWP : PackedWindowPtr; FName : string);
```

Purpose

Store the packed window pointed to by PWP in FName.

Comments

This routine allows you to store a packed window created by PackWindow as a binary file. If an error occurs, the error code is returned in CrtError. Besides the usual I/O error codes, the only possible value for CrtError is CrtNilPointer (meaning that PWP is nil). Please note that this routine will overwrite any existing file of the same name as FName without asking for confirmation.

Example

```
PWP := PackWindow(1, 1, 80, 25);
WritePackedWindow(PWP, 'WINDOW.DAT');
```

Save the entire screen as a packed window and store it in WINDOW.DAT.

See Also

PackWindow WritePackedWindow

C. TPWINDOW: Popup Windows

TPWINDOW manages any number of text windows displayed on the default video page of the physical screen. It also implements virtual screens which can be larger than the physical screen, up to 64K bytes of storage each. It supports stacked or tiled windows with frames and shadows, and has options for exploding windows.

Note that TPWINDOW and TPSCREEN perform similar functions in different ways. You should choose one or the other to use in your programs, but not both. Incorporating both units in the same program leads to conflicts, and is prohibited by special validation code in the initialization block of each unit. We recommend that you choose TPWINDOW over TPSCREEN if at all possible. TPSCREEN is provided primarily for compatibility with earlier versions of Turbo Professional.

In TPWINDOW's terms, a *window* is any rectangular portion of the screen whose properties are described by a window record. Each record holds information about the window's position, size, video attributes, cursor shape and position, frame, and header, as well as buffers storing the contents of the screen beneath the window and (optionally) the window itself. In the spirit of modular programming, however, TPWINDOW hides all these dirty details from you and instead exports a set of high-level functions to make window management as easy as possible. TPWINDOW offers routines to initialize and deallocate these records, to display and erase windows, and to manage stacked windows.

Once TPWINDOW displays a window, your program can write to it using any of the fast screen access methods of TPCRT. TPWINDOW defines a coordinate system relative to the upper left corner of the window. Provided that your program uses the Write and WriteLn procedures, text will automatically wrap and scroll within the window.

When TPWINDOW removes a window from the screen, it restores not only the screen contents uncovered by the window but also the cursor shape and position, the previous window coordinate system, and the default video attributes. Optionally, it will also save an image of the window just removed so that your application can redisplay it later without needing to reinitialize its contents.

TPWINDOW can act in this intelligent manner only by managing multiple windows on a *window stack*. Each window is pushed onto this stack whenever it is displayed and popped from the stack when it is erased. Thus, windows may be erased only in reverse order. TPWINDOW does, however, provide a special routine to "front" a window--that is, to move a window from any location in the stack to the top of the stack in one fast operation.

In this section we will use several terms that deserve clear definition. An *allocated* window is one whose window record has been allocated (on the heap) and initialized with default values for window position, attributes and so on. An allocated window may or may not be visible on the screen. An *active window* is one that is currently drawn on-screen, or one that is temporarily hidden by an overlapping window. The *current window* is the window most recently displayed. The window coordinate system as well as defaults for video attributes and the cursor shape are defined by the current window. Thus Write and WriteLn statements will always draw within the current window. An *active* window can become the *current* window in one of two ways: either by erasing windows until the desired window is at the top of the stack, or by calling the "fronting" routine (SetTopWindow) to pull the desired window to the top. TPWINDOW will not allow you to display an already active window a second time, since this would overwrite the window's screen buffers. However, TPWINDOW does provide routines which will either display a new window or pull an already active one to the top of the stack.

In some cases, TPWINDOW differentiates between "tiled" and "stacked" windows. A tiled window system is one where none of the windows overlap, whereas stacked windows have at least some degree of overlap. With tiled windows, TPWINDOW can switch between windows much more quickly because it doesn't need to restore any portion of the screen during the switch. Instead it can simply move the cursor, reset the window coordinates, and change the default text attribute. TPWINDOW 'an't tell whether you're using tiled or stacked windows. It is therefore your esponsibility to call the proper routine when you want to select a different window. Note, however, that it's always safe to use the routines for stacked windows.

Like TPSCREEN, TPWINDOW offers a color mapping facility. By using it, your application can define a single set of video attributes that will look equally good on both color and monochrome monitors. (The routine used to perform color mapping, MapColor, is actually located in the TPCRT unit, however, since it is also used by TPMENU.)

For good examples of how to use the principal routines in TPWINDOW, you should examine the sample program WINWOW.PAS. This program demonstrates both the speed and convenience of TPWINDOW's window management.

The following routines are declared in the interface section of TPWINDOW for the primary use of the TPMENU unit, which must access internal functions that are not needed by normal applications. The description section following does not contain entries for:

```
procedure ExplodeFrame(W : WinDrawPtr);
  {-Explode a window frame}

procedure ImplodeFrame(W : WinDrawPtr);
  {-Erase an exploding window from the screen}

procedure SaveShadowFrame(W : WinDrawPtr);
  {-Save the screen region under the window shadow}

procedure DrawShadowFrame(W : WinDrawPtr);
  {-Shadow a window by changing the attributes of underlying text}

procedure RestoreShadowFrame(W : WinDrawPtr);
  {-Restore screen region under shadow}
```

TPWINDOW: Declarations

Types

```
VScreen =
  record
    VRows : Word;          {Rows in the screen}
    VCols : Word;          {Columns in the screen}
    VSeg : Word;           {Segment where it's located}
    VEnd : Word;           {One past last valid offset in segment}
    VPtr : BufP;           {Allocated buffer}
  end;
```

Virtual screens managed by TPWINDOW are of type VScreen. The bulk of the memory usage of a virtual screen is on the heap, pointed to by the field VPtr. A virtual screen can be any size up to the limits of a 64K segment. Remember that each row,column position of a virtual screen uses 2 bytes of storage, one for the character and another for the video attribute. An application should not need to refer to the fields of a VScreen record--TPWINDOW provides a number of routines for manipulating virtual screens.

```
WindowPtr = Pointer;
```

Each Turbo Professional window variable should be given the type WindowPtr. A window consumes only the size of a pointer (four bytes) in the segment where the window variable is declared. After a call to MakeWindow, the initialized WindowPtr points to a variably sized region on the heap (about 70 bytes plus one or two screen buffers), which contains all the information TPWINDOW needs to manage the window. Although TPWINDOW exports only this opaque pointer type, you may wish to refer to WindowRec in the interface section of the unit to see exactly what a window record contains.

```
ShadowType = (SmallShadow, BigShadow);
```

Defines how TPWINDOW draws shadows around windows. BigShadow creates a see-through shadow, one character high and two characters wide. SmallShadow creates an opaque shadow, one-half character high and one character wide.

Constants

```
Explode : Boolean = False;
```

Set this value True prior to calling MakeWindow for any window that should "explode" when displayed and erased. MakeWindow stores the current value of Explode within the window record and uses it for subsequent displays of that window. The frame and background color of an exploding window appear or disappear in stages, leading to a dynamic effect that is appropriate for some applications.

```
ExplodeDelay : Word = 15;
```

This value determines the delay in milliseconds between stages of an explosion. Adjust it for the desired effect. Like the other global constants in TPWINDOW, MakeWindow stores the current value of ExplodeDelay when a window is defined.

```
Shadow : Boolean = False;
```

Set this value True prior to calling MakeWindow for any window that should display a shadow. MakeWindow stores the current value of Shadow within the

window record and uses it for subsequent displays of that window. In Turbo Professional, shadows always appear to the right and below any window.

```
ShadowAttr : Byte = $07;
```

When TPWINDOW displays a shadowed window in BigShadow mode, it changes the video attribute of the shadowed region to this attribute. By choosing an appropriate attribute, you can still read any text underlying the shadow. If you'd like to totally obliterate underlying text, use a ShadowAttr value with equal background and foreground colors. For example, the value $77 sets the shadow region to solid dim white. Set ShadowAttr prior to calling MakeWindow and it will store that value in the window record for subsequent display.

```
SoundFlagW : Boolean = True;
```

Exploding windows will also make noise if SoundFlagW is true. Note that when exploding windows are activated, sound is activated by default. You'll need to try the sound effects to decide whether they're appropriate for a given application.

```
ShadowMode : ShadowType = BigShadow;
```

Specifies the kind of shadow to use when shadows are enabled. This value is evaluated during a call to MakeWindow and stored in each window record thereafter.

Variables

```
CurrentWindow : WindowPtr;
```

This variable always points to the *current* window. If no windows are on-screen, CurrentWindow is nil.

ActivateTiledWindow

Declaration

```
function ActivateTiledWindow(W : WindowPtr) : Boolean;
```

Purpose

Display or reselect tiled window.

Comments

Use this routine to display a tiled window when you're not sure if that window is already on the screen. ActivateTiledWindow calls either DisplayWindow (if the window is not active) or SetTopTiledWindow (if the window is active).

Examples

```
Status := ActivateTiledWindow(A);
Status := ActivateTiledWindow(B);
Status := ActivateTiledWindow(A);
```

The first call to ActivateTiledWindow opens a new window A. The second call with window A as a parameter doesn't fully redraw the window but just repositions the cursor and resets the window coordinates.

See Also

DisplayWindow

SetTopTiledWindow

Declaration

```
procedure ActivateVScreen(V : VScreen);
```

Purpose

Switch fastwriting to the specified virtual screen.

Comments

When you call this routine, certain of TPCRT's fastwriting routines are rerouted so that their output goes to the specified virtual screen instead of the physical screen. The following TPCRT routines are affected:

```
FastWrite        FastText        FastVert        FastFill
FastRead         ReadAttribute   ChangeAttribute FlexWrite
FrameWindow
```

Do NOT attempt to use the following calls to write to the virtual screen:

```
Write            WriteLn
FastWriteWindow  FastTextWindow  FastVertWindow  FastFillWindow
FastCenter       FastFlush       FastReadWindow  ReadAttributeWindow
ChangeAttributeWindow            FlexWriteWindow FastWriteClip
```

Write and WriteLn statements will continue to write to the physical screen. The remaining calls will write to the virtual screen, but they will use window coordinates associated with the physical screen.

TPWINDOW's virtual screens do not support window coordinates within the virtual screen itself. Hence, the row and column coordinates specified to one of the FastWrite routines are always absolute with respect to the upper left corner of the virtual screen.

The FastWrite position coordinates are specified using Byte variables. As a result, the largest valid coordinate in either dimension is 255. To avoid subtle bugs, ActivateVScreen will refuse to switch writing to the virtual screen if either dimension of the specified screen exceeds 255. In this event, subsequent calls to FastWrite will continue to appear on the physical screen. TPWINDOW's WriteVScreen procedure may be used to write to a virtual screen with coordinates exceeding 255.

Be careful not to write a string that runs off the lower right hand corner of a virtual screen. When you do this on the physical screen, no particular harm is done. On a virtual screen, however, the overrun wipes out some other portion of the Pascal heap, with possibly disastrous consequences. The FastWrite routines do not protect against this error, although the WriteVScreen procedure does.

Point the fastwriting routines back to the physical screen by calling DeactivateVScreen. Copy the virtual screen to the physical screen by calling MoveVScreenToWindow.

ActivateVScreen

Example

```
var
  V : VScreen;
begin
  if not MakeVScreen(V, 100, 100) then begin
    WriteLn('Insufficient memory');
    Halt;
  end;
  ClearVScreen(V, $1E, ' ');
  ActivateVScreen(V);
  FastWrite('Hello virtual screen', 1, 1, $1E);
end.
```

The FastWrite does not appear immediately on the physical screen, but is written to row 1 column 1 of the virtual screen V.

See Also

DeactivateVScreen
FastWrite (TPCRT)
MoveVScreenToWindow
WriteVScreen

Declaration

```
function ActivateWindow(W : WindowPtr) : Boolean;
```

Purpose

Display or reselect stacked window.

Comments

Use this routine to display a stacked window when you're not sure if that window is already on the screen. ActivateWindow calls either DisplayWindow (if the window is not active) or SetTopWindow (if the window is active).

Examples

```
Status := ActivateWindow(A);
Status := ActivateWindow(B);
Status := ActivateWindow(A);
```

The first call to ActivateWindow opens a new window A. The second call restacks the windows so that window A is on top and current.

See Also

DisplayWindow SetTopWindow

Declaration

```
procedure ClearVScreen(V : VScreen; Attr : Byte; Fill : Char);
```

Purpose

Clear a virtual screen with specified attribute and character.

Comments

This routine takes the place of the ClrScr routine for virtual screens. Unlike the normal ClrScr routine, with this one you specify the video attribute (Attr) and fill character (Fill) for the cleared screen as parameters. Typically, the fill character will be a space. This routine always clears the entire virtual screen.

Since the MakeVScreen routine does not clear a newly allocated screen, you should typically call ClearVScreen immediately after MakeVScreen.

Examples

```
var V : VScreen;
...
if not MakeVScreen(V, 100, 100) then begin
  WriteLn('Insufficient memory');
  Halt;
end;
ClearVScreen(V, $1E, ' ');
```

Clears the virtual screen to all spaces, in yellow on blue.

See Also

MakeVScreen

DeactivateVScreen

Declaration

```
procedure DeactivateVScreen;
```

Purpose

Restore fastwriting to the physical screen.

Comments

Call this routine after previously calling ActivateVScreen. Fastwriting then returns to the physical screen.

See ActivateVScreen for further discussion, including a list of fastwriting routines affected by this call.

Examples

```
var
  V : VScreen;
begin
  if not MakeVScreen(V, 100, 100) then begin
    WriteLn('Insufficient memory');
    Halt;
  end;
  ClearVScreen(V, $1E, ' ');
  ActivateVScreen(V);
  FastWrite('Hello virtual screen', 1, 1, $1E);
  DeactivateVScreen;
  FastWrite('Hello physical screen', 1, 1, $1E);
end.
```

Writes first to the virtual screen and then to the physical.

See Also

ActivateVScreen

Declaration

```
function DisplayWindow(W : WindowPtr) : Boolean;
```

Purpose

Display the specified window, returning True if successful.

Comments

The parameter W specifies a window previously allocated and initialized by a call to MakeWindow. DisplayWindow stores the screen region about to be covered, draws the specified window (optionally clearing it or restoring its previous contents), initializes the cursor shape and position, sets the default video attribute for further writing, pushes the window record onto the active stack, and assigns the new window to the CurrentWindow variable.

If the Shadow or Explode variables were set True when the window W was first defined with MakeWindow, DisplayWindow will do what's necessary to create explosions and shadows.

DisplayWindow will move the cursor to the upper left corner of the window unless the SaveWindow flag was set by MakeWindow. If SaveWindow was set and the window has been displayed at least once, DisplayWindow will restore the window's contents and move the cursor to the position it had when the window was last erased.

DisplayWindow will return False if the specified window is already active, or if insufficient heap space is available to add a stack record (which requires 8 bytes).

Example

```
var Main : WindowPtr;
...
if not MakeWindow(Main, 1, 1, 80, 25, True, True, False,
                  WAttr, FAttr, HAttr, 'Main Window') then {Error};
...
if not DisplayWindow(Main) then {Error};
```

MakeWindow initializes the window variable Main. DisplayWindow then displays Main, which covers the entire screen. Main has a frame in video attribute FAttr with the header 'Main Window' shown in attribute HAttr. Because ClearWindow was True and SaveWindow was False during the call to MakeWindow, DisplayWindow will clear the window using attribute WAttr, and will move the cursor to the upper left corner of the window.

See Also

EraseTopWindow MakeWindow

DisposeVScreen / DisposeWindow

Declaration

```
procedure DisposeVScreen(var V : VScreen);
```

Purpose

Deallocate space for virtual screen.

Comments

This routine disposes of all heap space allocated for the virtual screen V by MakeVScreen. Note that you cannot use the virtual screen V after making this call, since DisposeVScreen fills all fields of the record V with zeros.

Examples

```
var V : VScreen;
...
if not MakeVScreen(V, 100, 100) then begin
  WriteLn('Insufficient memory');
  Halt;
end;
ClearVScreen(V, $1E, ' ');
ActivateVScreen(V);
FastWrite('Hello virtual screen', 1, 1, $1E);
DeactivateVScreen;
FastWrite('Hello physical screen', 1, 1, $1E);
DisposeVScreen(V);
```

See Also

MakeVScreen

Declaration

```
procedure DisposeWindow(W : WindowPtr);
```

Purpose

Deallocate heap space for specified window.

Comments

When a window is no longer needed, call DisposeWindow to deallocate the heap space it used. No such call is required at the end of a program since DOS automatically reclaims all the memory used by the program.

Example

```
if not MakeWindow(Main, 1, 1, 80, 25, True, True, False,
                  WAttr, FAttr, HAttr, 'Main Window') then {Error};
...
DisposeWindow(Main);
```

Main uses about 4070 bytes of heap space (70 bytes for the window record plus 4000 bytes for a single screen buffer). DisposeWindow reclaims this heap space.

See Also

MakeWindow

Declaration

```
function EraseTopWindow : WindowPtr;
```

Purpose

Erase the most recently displayed window, returning a pointer to it.

Comments

Windows are removed from the screen in the order exactly opposite to that when they were displayed, hence the name of this function.

When a window is erased, EraseTopWindow performs several actions. If the window has the SaveWindow flag set, the current window contents and the cursor position and shape are stored before the window is erased. Then the underlying contents of the screen are restored. Finally EraseTopWindow restores the cursor, window coordinates, and text attributes of the underlying window.

When EraseTopWindow erases the current window, it returns a pointer to the window it just erased. The pointer is particularly handy for calling DisposeWindow to reclaim the window's heap space (if the window is longer needed within the program). Calling EraseTopWindow is harmless when no windows are displayed--the function then does nothing but return a nil pointer.

Examples

```
DisposeWindow(EraseTopWindow);
```

Erases the current window and deallocates its heap space.

```
repeat
  W := EraseTopWindow;
until W = nil;
```

Erases all active windows, but leaves them allocated for later use.

See Also

DisposeWindow

FastWriteClip

Declaration

```
procedure FastWriteClip(S : string; Row, Col, Attr : Byte);
```

Purpose

Write string S in current window, using window-relative coordinates, clipping at right edge.

Comments

FastWriteClip is a shell around FastWriteWindow from the TPCRT unit. Besides using window-relative coordinates, FastWriteClip also truncates the displayed string to keep it from extending past the window boundaries. FastWriteClip doesn't wrap at window boundaries, as occurs with Write and WriteLn statements.

Example

```
Status := MakeWindow(W, 1, 1, 10, 2, True, True, False, WA, FA, HA, '');
Status := DisplayWindow(W);
FastWriteClip('a long sentence in a short window', 1, 1, WA);
```

Since the current window has room for only 8 characters of text, FastWriteClip truncates the string so that only 'a long s' shows in the window.

See Also

FastWriteWindow (TPCRT)

Declaration

```
procedure FastWriteStack(W : WindowPtr; S : string; Row, Col, Attr : Byte);
```

Purpose

Write to specified window, whether or not it's obscured.

Comments

This routine works like FastWriteWindow with one important enhancement. It allows writing to any active window, even when that window is partially or fully obscured by windows stacked on top of it.

W specifies the window to be written to. The window must be active, or FastWriteStack exits immediately. If the window is the current one, FastWriteStack simply transfers control to FastWriteWindow. Otherwise, FastWriteStack uses virtual screen techniques to temporarily unstack the overlapping windows, write the string, restack the windows, and transfer the result back to the physical screen.

As a result of this activity, FastWriteStack is not very fast, especially when there are many windows on top of the specified one. Even so, it produces some very impressive effects when properly used.

Row, Col, and Attr are interpreted exactly as for FastWriteWindow.

In order for FastWriteStack to work properly, all active windows must have the SaveWindow flag set True. (SaveWindow is the last boolean parameter specified in a call to MakeWindow.)

FastWriteStack does not clip strings that extend beyond window boundaries.

Examples

```
var
  OK : Boolean;
  W : WindowPtr;
  X : WindowPtr;
  R : Byte;
begin
  OK := MakeWindow(W, 1, 1, 60, 25, True, True, True, $1E, $17, 0, '');
  OK := MakeWindow(X, 20, 5, 40, 21, True, True, True, $3F, $30, 0, '');
  OK := DisplayWindow(W);
  OK := DisplayWindow(X);
  for R := 1 to 23 do
    FastWriteStack(W, CharStr('A', 58), R, 1, $1E);
end.
```

Window X is a small window, displayed on top of a larger window W in the background. FastWriteStack writes to the background window while leaving X undisturbed.

See Also

FastWriteClip

FastWriteWindow (TPCRT)

KillTiledWindow / KillWindow

Declaration

```
procedure KillTiledWindow(var W : WindowPtr);
```

Purpose

Erase and dispose of a tiled window, no matter where it is.

Comments

If the window W is active, KillTiledWindow finds it in the window stack, erases it, and disposes of it. If W isn't active, KillTiledWindow just disposes of it. The pointer W is set to nil after the call.

This call is valid only for a tiled window system. If there is any overlap between active windows, the screen will be corrupted after calling KillTiledWindow. Use KillWindow for overlapping windows.

Declaration

```
procedure KillWindow(var W : WindowPtr);
```

Purpose

Erase and dispose of a window, no matter where it is.

Comments

If the window W is active, KillWindow finds it in the window stack, erases it, and disposes of it. If W isn't active, KillWindow just disposes of it. The pointer W is set to nil after the call.

Since KillWindow may call SetTopWindow, it is important that the SaveWindow flag (specified in the call to MakeWindow) is True for any windows that may be involved.

See Also

EraseTopWindow

Declaration

```
function MakeVScreen(var V : VScreen; Rows, Cols : Word) : Boolean;
```

Purpose

Return True after allocating a virtual screen.

Comments

This function initializes a virtual screen record for further use. It first validates the number of Rows and Cols in the virtual screen, assuring there is at least one of each, and that the overall size of the virtual screen buffer does not exceed 64K bytes. Then it allocates heap space for the virtual screen. If any of these fail, MakeVScreen returns False, otherwise it returns True.

Never attempt to write to a virtual screen prior to calling MakeVScreen.

If either Rows or Cols exceeds 255, the ActivateVScreen routine will not work for the virtual screen, since the TPCRT fastwriting routines do not allow row and column coordinates that large. The WriteVScreen routine is still available for these cases, however.

Calling MakeVScreen does not clear the virtual screen. Hence, this call should typically be followed by a call to ClearVScreen.

Examples

```
Status := MakeVScreen(V1, 100, 100);
Status := MakeVScreen(V2, 256, 100);
Status := MakeVScreen(V3, 255, 255);
Status := MakeVScreen(V4, 409, 80);
```

Screen V1 consumes about 20000 bytes on the heap, and FastWrite will write to it after ActivateVScreen is called. Screen V2 consumes about 51200 bytes on the heap, and can be written to only by using WriteVScreen. Screen V3 cannot be allocated because it would require 130050 bytes of heap space. Screen V4 is the longest 80 column screen possible, consuming about 65520 bytes of heap space.

See Also

ActivateVScreen

ClearVScreen

MakeWindow

Declaration

```
function MakeWindow(var W : WindowPtr;
                    XLow, YLow : Byte;
                    XHigh, YHigh : Byte;
                    DrawFrame : Boolean;
                    ClearWindow : Boolean;
                    SaveWindow : Boolean;
                    WindowAttr : Byte;
                    FrameAttr : Byte;
                    HeaderAttr : Byte;
                    Header : string
                    ) : Boolean;
```

Purpose

Allocate and initialize, but do not display, a new window.

Comments

MakeWindow defines the properties of each new window. Call it to allocate and initialize a window variable before it must be displayed.

The parameter W is the window variable to be initialized. Upon return, it will hold a pointer to a structure that contains all needed information about the window.

XLow, YLow, XHigh, and YHigh define the upper left and lower right corners of the window. Valid X coordinates range from 1 to ScreenWidth and valid Y coordinates range from 1 to ScreenHeight. Be warned that MakeWindow does not validate the coordinate values in any way. If the window is framed, the window coordinates include the positions used by the frame. When a framed window is actually displayed, though, the relative coordinate system will define position 1,1 just inside the upper left hand corner of the frame.

Set the DrawFrame parameter True if you want TPWINDOW to draw a frame around the window when it is displayed. The characters used for the frame are those current in the TPCRT variable FrameChars when MakeWindow is called. You may specify a desired set of frame characters by directly assigning them to FrameChars or by calling TPCRT's procedure SetFrameChars. The frame will be drawn in the video attribute specified by FrameAttr.

Set the ClearWindow parameter True if you want TPWINDOW to clear the body of the window when it is displayed. The window will be filled with blanks in video attribute WindowAttr.

By setting SaveWindow to True you tell TPWINDOW to save the current contents of this window before it is erased. The next time the window is displayed, TPWINDOW will restore its previous contents. The SaveWindow parameter, when True, overrides the ClearWindow flag except for the very first time the window is displayed.

If you want to center a title string on the top bar of the window frame, specify a string for the Header parameter. Otherwise, pass an empty string in this position. The header string will be displayed in the video attribute HeaderAttr.

The initial cursor shape for a window will be the one active when MakeWindow is

called. Thus if the cursor is hidden when a new window is allocated, the default cursor for that window will also be hidden. When the window is active, the cursor may be changed using any of TPCRT's routines for that purpose.

MakeWindow allocates heap space for the window record and for required screen buffers. The window record itself requires about 70 bytes. Screen buffer size is based on the window coordinates:

BufferSize = 2*(XHigh-XLow+1)*(YHigh-YLow+1) bytes

If the SaveWindow parameter is True, two buffers of this size are allocated, otherwise a single one is. MakeWindow allocates additional heap space to hold the Header string, if any. MakeWindow will return false if it is unable to allocate these heap blocks.

MakeWindow also takes account of several global typed constants by storing their current values in the window record. The constants Shadow, ShadowAttr, ShadowMode, Explode, ExplodeDelay, and SoundFlagW all affect the eventual display of the window. Be sure that these constants are in the desired state prior to calling MakeWindow.

Note that MakeWindow does not by itself display a window. To display the window, use DisplayWindow. To deallocate a window, use DisposeWindow.

Examples

```
var Main, Status : WindowPtr;
...
if not MakeWindow(Main, 1, 1, 80, 25, True, True, False,
                  WAttr, FAttr, HAttr, 'Main Window') then {Error};
```

Allocates a full-screen window which will be framed and initially cleared, but not saved. The window title will be 'Main Window'.

```
if not MakeWindow(Status, 10, 10, 30, 15, True, True, True,
                  WAttr, FAttr, HAttr, 'Status') then {Error};
```

Allocates a small status window. Since the SaveWindow flag is True, the contents of this window will be saved and restored when the window is erased and redisplayed.

See Also

Declarations (TPCRT)

DisposeWindow

MoveVScreenToWindow

Declaration

```
procedure MoveVScreenToWindow(V : VScreen; Row, Col : Word);
```

Purpose

Copy contents of virtual screen at Row,Col to current window.

Comments

Use this routine to copy some or all of a virtual screen to the physical screen, thus making the contents of the virtual screen visible.

The upper left corner of the region to be copied is at coordinate Row,Col within the virtual screen. MoveVScreenToWindow takes a portion of the virtual screen the size of the currently active window and copies it into the current window on the physical screen. If the current window is larger than the available virtual screen space starting at Row,Col, MoveVScreenToWindow copies the available portion of the virtual screen to the window, and leaves the rest of the window unchanged.

MoveVScreenToWindow uses the TPCRT variables WindMin and WindMax to determine the current window coordinates. These variables are set by calls to TPCRT's Window procedure and are also managed by TPWINDOW's various windowing routines. Note that the coordinates in WindMin and WindMax do not include the frames or shadows of windows managed by TPWINDOW. Therefore, MoveVScreenToWindow will fill only the central portion of a window, and won't affect the frame.

By varying the Row,Col parameters passed to MoveVScreenToWindow, you can achieve the effect of horizontal, vertical or diagonal scrolling on the physical screen. See the WINWOW demo program for an example.

To copy from the physical screen to a virtual screen, call MoveWindowToVScreen.

Examples

```
var
  V : VScreen;
  S : string;
  C : Word;
begin
  {Initialize and clear a one row virtual screen, 255 cols wide}
  if not MakeVScreen(V, 1, 255) then begin
    WriteLn('Insufficient memory');
    Halt;
  end;
  ClearVScreen(V, $07, ' ');

  {Write a long string to the virtual screen}
  S := '';
  while Length(S) < 255 do
    S := S+'TPWINDOW virtual screens ';
  WriteVScreen(V, S, 1, 1, $07);

  {Set up a smaller window on physical screen}
  ClrScr;
  Window(1, 1, 40, 1);
```

```
    {Scroll the virtual screen over the physical window}
    for C := 1 to 250 do begin
      MoveVScreenToWindow(V, 1, C);
      Delay(20);
    end;
  end.
```

Simulates horizontal scrolling by using MoveVScreenToWindow to copy a large virtual screen to a smaller window.

See Also

MoveWindowToVScreen

MoveWindow

Declaration

```
function MoveWindow(XDelta, YDelta : Integer) : Boolean;
```

Purpose

Move current window by specified distance. Positive values indicate movement to the right or down.

Comments

Relocate the current window with this routine. The window will remain the same size, but its entire contents will be moved to a different location on the screen.

XDelta and YDelta specify how much to move the window. Use negative numbers to move the window to the left or up, and positive numbers to move right or down. It is your responsibility to assure that the shift still allows the entire window to fit on the screen. It it won't, MoveWindow will exit without redrawing and return a False function result.

MoveWindow uses virtual screen techniques to avoid unsightly screen flashing while the window moves. In order to do so, it must temporarily allocate a buffer on the heap. The buffer is as large as a rectangle which surrounds the window in its before and after positions. The largest possible buffer for a 25 line screen is 25*80*2 = 4000 bytes. If MoveWindow cannot allocate the buffer, it returns a False function result.

Once the window contents have been shifted and redrawn, MoveWindow adjusts the current window coordinates and the cursor position. It returns True to indicate success.

Examples

```
Status := MakeWindow(W, 1, 1, 10, 5, True, True, False, WA, FA, HA, '');
Status := DisplayWindow(W);
Write('hello');
Status := MoveWindow(1, 0);
Status := MoveWindow(0, 1);
Status := MoveWindow(-1, -1);
```

Here we define a small window, display it, move it one column to the right, move it one row down, then restore it to its original position.

The WINWOW demonstration program shows off TPWINDOW's ability to move and resize windows quickly.

See Also

ResizeWindow

Declaration

```
procedure MoveWindowToVScreen(V : VScreen; Row, Col : Word);
```

Purpose

Copy contents of current window to virtual screen at Row,Col.

Comments

This routine performs the opposite action to that of the MoveVScreenToWindow procedure. It copies the contents of the current window on the physical screen to a virtual screen, starting at coordinate Row,Col of the virtual screen.

If the current window is larger than the available virtual screen space starting at Row,Col, MoveWindowToVScreen copies the portion of the window that will fit.

MoveWindowToVScreen uses the TPCRT variables WindMin and WindMax to determine the current window coordinates. These variables are set by calls to TPCRT's Window procedure and are also managed by TPWINDOW's various windowing routines. Note that the coordinates in WindMin and WindMax do not include the frames or shadows of windows managed by TPWINDOW. Therefore, MoveWindowToVScreen will copy only the central portion of a window, and won't capture the frame.

Examples

```
var
  V : VScreen;
begin
  if not MakeVScreen(V, 25, 80) then begin
    WriteLn('Insufficient memory');
    Halt;
  end;

  {Copy the entire physical screen to the virtual screen}
  Window(1, 1, 80, 25);
  MoveWindowToVScreen(V, 1, 1);

...

  {Restore the physical screen}
  MoveVScreenToWindow(V, 1, 1);
end.
```

Allocates a virtual screen the same size as the physical screen, then copies the entire physical screen to the virtual. At that point, the physical screen may be changed, and later restored by the MoveVScreenToWindow call, or the virtual screen may be modified, and displayed in one quick flash with the final call.

See Also

MoveVScreenToWindow

ReadVScreen

Declaration

```
procedure ReadVScreen(V : VScreen; Number : Byte;
                      Row, Col : Word; var S : string);
```

Purpose

Read Number characters from the virtual screen starting at Row,Col.

Comments

This routine reads a string directly from a virtual screen. It is the equivalent of FastRead for virtual screens. It may be used instead of ActivateVScreen and FastRead. Besides specifying the virtual screen to read from, the parameters to ReadVScreen are defined just like those to FastRead (with the exception that Row and Col are Words rather than Bytes).

See WriteVScreen for a discussion of the pros and cons of this technique as compared to ActivateVScreen.

Examples

```
var
  V : VScreen;
  S : string;
begin
  if not MakeVScreen(V, 100, 100) then begin
    WriteLn('Insufficient memory');
    Halt;
  end;
  ClearVScreen(V, $1E, ' ');
  WriteVScreen(V, 'Hello virtual screen', 1, 1, $1E);
  ReadVScreen(V, 20, 1, 1, S);
end.
```

The string S ends up containing 'Hello virtual screen'.

See Also

ActivateVScreen
FastRead (TPCRT)
WriteVScreen

Declaration

```
function ResizeWindow(XDelta, YDelta : Integer; Fill : Char) : Boolean;
```

Purpose

Resize current window. Positive values indicate growth to the right or down.

Comments

The upper left corner of the window will remain in the same position, but the right hand and/or lower edges will move to reflect the new size.

XDelta and YDelta specify how much to resize the window. Use negative numbers to shrink the window, and positive numbers to increase its size. It is your responsibility to assure that the new window still fits entirely on the screen. If it won't, ResizeWindow will exit without redrawing and return a False function result.

When the window grows, ResizeWindow fills in the newly created sections of the window with the Fill character that you specify. Typically, Fill will be a space character.

ResizeWindow uses virtual screen techniques to avoid unsightly screen flashing while the window is changed. In order to do so, it must temporarily allocate a buffer on the heap that is big enough to hold the window, before or after the resizing. The largest possible buffer for a 25 line screen is 25*80*2 = 4000 bytes. If ResizeWindow cannot allocate the buffer, it returns a False function result.

Once resizing is done, ResizeWindow adjusts the current window coordinates and pulls the cursor back into the window if needed. It returns True to indicate success.

Resizing a window in many small steps tends to fragment the heap, because TPWINDOW must deallocate and allocate window buffers to accommodate the new size. To minimize fragmentation, TPWINDOW changes the size of the window buffers by multiples of two, thus reducing the number of times it must reallocate.

Examples

```
Status := MakeWindow(W, 1, 1, 10, 5, True, True, False, WA, FA, HA, '');
Status := DisplayWindow(W);
Write('hello');
Status := ResizeWindow(0, 15, ' ');
Status := ResizeWindow(0, 15, ' ');
```

The first call to ResizeWindow will successfully increase the window height from 5 rows (1..5) to 20 rows (1..20). The second call to ResizeWindow will fail, returning a Status value of False, since the 35 line window won't fit on the normal screen.

The WINWOW demonstration program shows off TPWINDOW's ability to move and resize windows quickly.

See Also

MoveWindow

ScrollWindow / SetInactiveFrame

Declaration

```
procedure ScrollWindow(Up : Boolean; Lines : Byte);
```

Purpose

Scroll current window Up or down (Up = False) the designated number of Lines.

Comments

ScrollWindow passes the coordinates for the active window, along with the Lines parameter, to ScrollWindowUp (if Up=True) or ScrollWindowDown (Up=False).

Example

```
ScrollWindow(True, 1);
```

In an 80 x 25 window, this would move rows 2-25 to rows 1-24 and clear row 25.

See Also

ScrollWindowDown (TPCRT) ScrollWindowUp (TPCRT)

Declaration

```
procedure SetInactiveFrame(W : WindowPtr; Fr : FrameArray;
                           FrameAttr, HeaderAttr : Byte);
```

Purpose

Define the frame type and attribute for windows that are visible but not current.

Comments

TPWINDOW provides the automatic capability to draw different frames for a window when it is current as opposed to on-screen but not current. By default, the two frames are equivalent. Call the SetInactiveFrame routine to establish different frame characters and colors for active non-current windows. Once this is done, TPWINDOW automatically switches to the "inactive" attributes when another window is displayed or pulled to the top.

Examples

```
FrameChars := #201#200#187#188#205#186;
Status := MakeWindow(A, 01, 1, 40, 25, True, True, False, WA, $0F, $0F, '');
Status := MakeWindow(B, 41, 1, 80, 25, True, True, False, WA, $0F, $0F, '');
SetInactiveFrame(A, #218#192#191#217#196#179, $07, $07);
SetInactiveFrame(B, #218#192#191#217#196#179, $07, $07);
Status := DisplayWindow(A);
Status := DisplayWindow(B);
```

The current window is displayed with a double bar frame and $0F for the frame color. Active but non-current windows are displayed with a single bar frame and frame attribute of $07. When this sequence of code is done, window A will have a single bar frame, and window B a double bar frame.

See Also

DisplayWindow

Declaration

```
function SetTopTiledWindow(W : WindowPtr) : Boolean;
```

Purpose

Make an already active, tiled window the current one.

Comments

This routine is applicable only when two or more tiled (non-overlapping) windows are currently displayed. In that situation, calling SetTopTiledWindow makes the specified window W the current window, and adjusts the window coordinates, cursor position, and default text color accordingly.

SetTopTiledWindow will return False if the specified window is not currently displayed. Since SetTopTiledWindow needs a small amount of heap space while it rearranges the window stack, it may also return False if there is insufficient free heap space.

This routine is much faster than SetTopWindow, which must use a virtual screen to erase and redraw the possibly overlapping windows.

Examples

```
Status := MakeWindow(A, 01, 1, 40, 25, True, True, False, WA, FA, HA, '');
Status := MakeWindow(B, 41, 1, 80, 25, True, True, False, WA, FA, HA, '');
Status := DisplayWindow(A);
Status := DisplayWindow(B);
Status := SetTopTiledWindow(A);
```

Here we define two windows that cover the entire screen without overlapping, display both of them, and then switch control back to the first window.

See Also

ActivateTiledWindow

SetTopWindow

SetTopWindow

Declaration

```
function SetTopWindow(W : WindowPtr) : Boolean;
```

Purpose

Move already active window to top of stack, returning True if successful.

Comments

A window manager that uses a stack has one inherent disadvantage: active windows that aren't current cannot be immediately accessed. The SetTopWindow function remedies that weakness by providing a method to make any active window the current one.

Specify the window you want to be current in the W parameter. This window must already be active, or SetTopWindow will do nothing but return a False function result. If the window is active, SetTopWindow will erase any windows down to and including the requested one and then redraw them in order with the requested one last (on top). If the requested window is already current, SetTopWindow does nothing but return a True result.

This strategy is viable only when the SaveWindow flag is True for all active windows. Otherwise, the restacked windows will lose their previous contents. SetTopWindow does *not* check to assure that the SaveWindow flag is indeed True for all windows.

Try the demonstration program WINWOW to see SetTopWindow in action.

Note that SetTopWindow uses virtual screen techniques in order to make all the intermediate steps of erasing and redrawing windows invisible. In order to do so, SetTopWindow must allocate a temporary heap buffer as large as the physical screen, typically 25*80*2 = 4000 bytes. If this amount of memory (plus a small additional amount for stack manipulation) is not available, SetTopWindow will return False.

Example

```
if not DisplayWindow(A) then {Error};
if not DisplayWindow(B) then {Error};
if not SetTopWindow(A) then {Error};
```

The SetTopWindow statement pulls window A from the bottom position on the stack to the top position.

See Also

MakeWindow
SetTopTiledWindow
WindowIsActive

Declaration

```
function VScreenCols(V : VScreen) : Word;
```

Purpose

Return number of columns in virtual screen.

Comments

This routine simply returns the VCols field from the virtual screen record. Applications should call this routine rather than accessing the virtual screen record directly in case the internal format of a virtual screen changes in the future.

Example

```
if not MakeVScreen(V, 25, 80) then begin
  WriteLn('Insufficient memory');
  Halt;
end;
WriteLn('Columns in screen V: ', VScreenCols(V));
```

Writes the value 80.

Declaration

```
function VScreenRows(V : VScreen) : Word;
```

Purpose

Return number of rows in virtual screen.

Comments

This routine simply returns the VRows field from the virtual screen record. Applications should call this routine rather than accessing the virtual screen record directly in case the internal format of a virtual screen changes in the future.

Example

```
if not MakeVScreen(V, 25, 80) then begin
  WriteLn('Insufficient memory');
  Halt;
end;
WriteLn('Rows in screen V: ', VScreenRows(V));
```

Writes the value 25.

WindowIsActive

Declaration

```
function WindowIsActive(W : WindowPtr) : Boolean;
```

Purpose

Return True if specified window is currently active.

Comments

This simple function may be useful for user-driven windowing systems. If a user requests that a particular window be displayed, the program can decide whether to use SetTopWindow or DisplayWindow depending on whether the requested window is already active.

Example

```
if WindowIsActive(W) then
  Status := SetTopWindow(W)
else
  Status := DisplayWindow(W);
```

This sequence avoids an illegal attempt to display the same window twice.

See Also

DisplayWindow

SetTopWindow

Declaration

```
procedure WriteVScreen(V : VScreen; S : string;
                       Row, Col : Word; Attr : Byte);
```

Purpose

Write a string to the specified virtual screen.

Comments

This routine writes a string directly to a virtual screen. It is the equivalent of FastWrite for virtual screens. It may be used instead of ActivateVScreen and FastWrite. Besides specifying the virtual screen to write to, the parameters to WriteVScreen are defined just like those to FastWrite (with the exception that Row and Col are Words rather than Bytes).

WriteVScreen has advantages and disadvantages relative to ActivateVScreen. On the positive side: WriteVScreen writes to the specified screen in a single call; it may be interspersed with normal WriteLn/FastWrite calls to the physical screen; it can write to virtual screens where one of the dimensions exceeds 255 rows or columns; it validates all writes to assure that the virtual screen buffer is not overrun. On the negative side: ActivateVScreen may be followed by calls to many routines besides Fastwrite, including FastText, FastFill, FlexWrite, and others; for a number of writes in a row, ActivateVScreen is more efficient.

Examples

```
var
  V : VScreen;
begin
  if not MakeVScreen(V, 100, 100) then begin
    WriteLn('Insufficient memory');
    Halt;
  end;
  ClearVScreen(V, $1E, ' ');
  WriteVScreen(V, 'Hello virtual screen', 1, 1, $1E);
end.
```

Writes a string to row 1, column 1 of the virtual screen.

See Also

ActivateVScreen

FastWrite (TPCRT)

D. TPMENU: Pulldown Menus

Fast, intuitive menus and windows are an expected feature of personal computer applications software today. As many of us know, polishing a program's user interface can take more time than writing the core of the application. TPMENU is a powerful collection of routines to reduce the time and drudgery of developing a sophisticated menu system.

Menus come in various styles. Borland's pulldown menus define one commonly used style. Lotus' single-line, horizontally scrolling menu is another. TPMENU easily emulates either of these styles, and lets you define others that fit your fancy. TPMENU also provides integral support for the latest techniques: shadowed windows, exploding frames, menu hot keys, and selection of items using a mouse.

TPMENU data structures are fully dynamic--the capacity of a given menu system is limited only by available heap space. TPMENU is implemented with a complex system of dynamic trees and lists. Fortunately, you won't need to worry about any of the details. TPMENU exports the high level routines that make it easy to use.

Perhaps the best way to introduce TPMENU is with a simple example:

```
program MenuDemo;
uses
  TpCrt,
  TpMenu;
var
  Ch : Char;
  Main : Menu;
  Key : MenuKey;

procedure InitMenu(var M : Menu);
const
  Color1 : MenuColorArray = ($0E, $2E, $03, $1E, $0B, $0E, $19, $78);
  Frame1 : FrameArray = #201#200#187#188#205#186;
  Frame2 : FrameArray = #203#200#203#188#205#186;
begin
  M := NewMenu([], nil);
  SubMenu(1,2,1,Horizontal,Frame1,Color1,'TPMENU');
    MenuItem('First',2,1,1,'');
    SubMenu(2,4,1,Vertical,Frame2,Color1,'');
      MenuItem('Alpha',1,1,2,'');
      MenuItem('Beta',2,1,3,'');
      PopSublevel;
    MenuItem('Second',12,1,4,'');
    SubMenu(11,4,1,Vertical,Frame2,Color1,'');
      MenuItem('Gamma',1,1,5,'');
      MenuItem('Delta',2,1,6,'');
      PopSublevel;
    PopSublevel;
  ResetMenu(M);
```

```
end;
begin
  InitMenu(Main);
  repeat
    Key := MenuChoice(Main, Ch);
    GoToXy(1, 25);
    Write('Selection=', Key, ' with keystroke code of ', Ord(Ch));
    Delay(1000);
    EraseMenu(Main, False);
  until (Ch = ^M) and (Key = 6);
end.
```

The procedure InitMenu initializes the menu system, which in this case is a simple two-level pulldown menu. The source code for InitMenu was generated by the supplied utility program MAKEMENU.

You'll note that InitMenu calls five TPMENU routines. NewMenu initializes a new, empty menu system and defines some of its global properties. SubMenu adds a new *submenu* (a rectangular area of the screen containing one or more menu items) to the current node of the menu system. Each submenu is defined by a position, orientation, frame type, set of colors, title and help row. MenuItem adds a new line item to the current submenu. A menu *item* is defined by its name, position within the submenu, keystroke used to select it, a code returned when the item is selected, and optional help string. PopSubLevel tells TPMENU that the current submenu is complete, and that initialization should continue at the previous level. ResetMenu assures that the initially selected item is the first of each submenu.

The main loop of the sample program calls MenuChoice, which is the central routine of TPMENU. MenuChoice displays the menu system, lets the user browse around it using the cursors, <Enter> and <Esc> keys, and returns a selection when done. The user's selection is returned in two parts: the keystroke used to select the item, and a unique code for the item selected. The keystroke might be needed to differentiate between a request for further help and a request to perform an action, or even to support "hotkey" access to a distant part of the menu system. The unique code is one assigned during menu initialization. Your program can decide which action to take based on this code.

One additional bit of information is available to help you determine what action to take. MenuCmdNum is a global variable of type MKStype that contains the last menu command executed by the user. (See the Declarations section, below.) In some cases, this value offers a more reliable means of determining what action was requested, most notably cases where TPMENU's mouse support is enabled, and cases where the user is given some means of changing key assignments (see Appendix B). In general, we therefore recommend using MenuCmdNum rather than the key code returned by MenuChoice.

When control returns from MenuChoice, the menu system will remain on-screen. The program can call TPMENU's EraseMenu routine at its discretion to restore the screen.

TPMENU also exports routines to read, write, and pack menu libraries on disk.

For a comprehensive example of how to use TPMENU, see the demonstration program MAKEMENU. MAKEMENU is a powerful tool that helps you interactively define hierarchical menu systems, and then writes the source code for you.

The following routines are declared in the interface section of TPMENU for the exclusive use of the MAKEMENU utility program, which needs access to internal functions that are not needed by normal applications. The description section following does not contain entries for:

```
procedure ClearWindow(XL, YL, XH, YH, Attr : Byte);
  {-Clear a region with specified attribute}

procedure DisposeSubMenu(var SubMnu : SubMenuP);
  {-Dispose of submenu and its children}

procedure DrawFrame(XL, YL, XH, YH, Attr : Byte; Frame : FrameArray);
  {-Draw a frame around a window}

procedure DrawItem(SubMnu : SubMenuP; Item : ItemP; UserFunc : Pointer);
  {-Draw one item in a submenu}

procedure DrawSubMenu(SubMnu : SubMenuP; UserFunc : Pointer);
  {-Draw a submenu on-screen}

procedure EraseSubMenu(SubMnu : SubMenuP);
  {-Erase a submenu from the screen}

procedure PopSubMenu(Mnu : Menu);
  {-Remove submenu from active stack}

procedure PushSubMenu(Mnu : Menu; SubMnu : SubMenuP);
  {-Put submenu onto active stack of the menu}
```

Constants

```
AllowPendingSubmenus: Boolean = False;
```

When this typed constant is True, the MenuChoice routine "remembers" whether to display submenus when the highlight bar is moved from item to item. The easiest way to describe this behavior is to use an example. Suppose a menu system starts with horizontal submenu that contains the items A, B, and C. Also suppose that items A and C each have vertical submenus containing some number of items, but that B has no submenu. After MenuChoice is called, assume that the user presses <Enter> while the highlight bar is over item A, making the vertical submenu appear. If the user presses the right arrow key once, the highlight bar moves to item B and the submenu disappears because B has no submenu. If the right arrow key is pressed once more, the effect of AllowPendingSubmenus becomes apparent. When it has the default value of False, C's submenu is not automatically displayed. When it is True, MenuChoice knows that a submenu was visible previously and automatically displays C's submenu.

```
FrameDelta : array[Boolean] of Byte = (1, 0);
```

This constant allows TPMENU to correct coordinates for menus with and without frames. Applications should not need to use it.

```
HideCursor : Boolean = True;
```

Set this constant False prior to calling MenuChoice if you want the blinking hardware cursor to remain visible while the menu is on-screen.

```
LotusFrame = #255#255#255#255#255#255;
NoFrame = LotusFrame;
```

Lotus-style menus are a special case for TPMENU because they have no frame bordering each menu. To denote this, the special FrameArray LotusFrame is defined. NoFrame is a synonym for LotusFrame. Applications using menus without frames should specify NoFrame wherever a FrameArray is called for.

```
MenuItemProc : Pointer = nil;
```

If this pointer is not nil, it must point to a non-nested procedure compiled under the FAR model with the following declaration:

```
procedure MyMenuItemProc(MKey : MenuKey);
```

This procedure is then called whenever the current item is changed (while MenuChoice or SelectMenuItem is active). By using the menu key for identification, the user routine can perform activities such as updating help or status information.

Notice the distinction between the user-supplied DisplayFunc (see NewMenu later in this section) and MenuItemProc routines. DisplayFunc is called for each and every item, current or not, each time it is displayed. MenuItemProc is called immediately after the current item of the current submenu is displayed.

The declaration for MenuItemProc does not provide information on which menu system is active. If an application contains more than one menu system, MenuItemProc should be changed prior to calling MenuChoice for each system.

```
MKSNone = 0;        {Not a command}
MKSAlpha = 1;       {An alphanumeric character}
MKSUp = 2;          {Cursor up}
MKSDown = 3;        {Cursor down}
MKSLeft = 6;        {Cursor left}
MKSRight = 7;       {Cursor right}
MKSExit = 8;        {Exit the menu}
MKSSelect = 9;      {Select the current item}
MKSHelp = 10;       {Provide help about current item}
MKSHome = 11;       {Cursor to first menu item}
MKSEnd = 12;        {Cursor to last menu item}
MKSProbe = 13;      {Attempt to select item with mouse}
MKSUser0 = 14;      {User-defined exit commands}
...
MKSUser3 = 17;
```

These are the codes for each of TPMENU's cursor movement and selection commands. Most of them are self explanatory. The MKSHelp command is activated whenever <F1> is pressed, or both mouse buttons are released. If a user-defined help routine has been specified, it is called at this time. The MKSProbe command was added specifically to support the mouse. This command is activated whenever the left mouse button is released. TPMENU's logic then checks to see whether the mouse cursor is aligned within a valid menu item, and if so, selects it.

The last four (MKSUser0..MKSUser3) are set aside to allow user-defined "exit commands" to be implemented. If one of these commands is executed, TPMENU will do the same things that it would if a menu hot key were pressed; it will then be up to you to check the global variable MenuCmdNum and perform the function associated with the command just issued. For more details on how to implement user-defined commands, see Appendix B.

```
NoHelp = 0;
```

Each menu in a TPMENU system may have a row of help text associated with it. If no help text is desired, specify NoHelp for the help row to minimize time and memory overhead.

```
SelectOnFirstMouseClick : Boolean = True;
```

When mouse support is enabled and this constant is True, TPMENU selects each menu item as soon as the mouse cursor is positioned over that item and the left button is pressed. When the constant is False, TPMENU does not reposition the menu highlight bar when the mouse button is pressed once. Selection occurs only when the mouse button is pressed a second time while the mouse cursor is over an already highlighted item.

Types

```
Menu = ^MenuRec;
```

A variable of this type is the main hook between your program and the menu system. TPMENU initializes a variable of type Menu when you call the NewMenu function, and continues to build upon the structure with succeeding calls to SubMenu and MenuItem. You then pass a variable of type Menu to the MenuChoice routine, which displays the menu system and allows the user to select an item. See TPMENU.IN0 for further details about the MenuRec type.

```
MenuCharSet = set of Char;
```

TPMENU exports this type to allow specification of a set of menu exit characters. Whenever the user presses one of the characters in this set, the MenuChoice function will exit and return the current menu selection to the caller.

```
MenuColorType =        {Colors used by the menu system}
(FrameColor,           {Color used for menu frames}
 HeaderColor,          {Color used for header string within frame}
 BodyColor,            {Color used for body of menu}
 SelectColor,          {Color used for the selected item}
 HiliteColor,          {Color used for the pick character of each item}
 HelpColor             {Color used for help row, if any}
 {$IFDEF Tpro5Menu}
 ,
 DisabledColor,        {Color used for temporarily disabled items}
 ShadowColor           {Color used for menu shadows}
 {$ENDIF}
 );

MenuColorArray = array[MenuColorType] of Byte;
```

MenuColorType defines the various colors used within a menu system. FrameColor is used for the box surrounding a menu. HeaderColor is for the menu title centered on the top bar of the frame. BodyColor defines the color for the body of the menu--all unselected items. SelectColor defines the color for the currently selected item in a menu. HiliteColor applies to the one character which may be used to select each item alphabetically. HelpColor is the attribute used for the optional help row. In Turbo Professional version 5, two additional attributes are defined: DisabledColor defines the color for menu items that are temporarily disabled, and hence cannot be selected. ShadowColor is the attribute to use for shadow boxes to the right and below a submenu. The shadow is selected by a separate mechanism, so ShadowColor will be ignored in some situations. The MenuColorArray is just a convenient way to collect these colors into a single structure.

```
MenuKey = LongInt;
```

Each menu selection is associated with a *key*. The key is a unique value which identifies the selection. TPMENU defines the key as a four byte data type. By use of typecasting, an application can store and retrieve enumerated types, integers, longints or pointers by using the key. If identification information will not fit in four bytes, a pointer to a larger data structure should be stored in the MenuKey. Generally, we recommend that the MenuKey value assigned to each item should

be a small integer, ranging from 1 to 10000. In this way, the MenuKey values are compatible with the help indexes used by the TPHELP unit. The MAKEMENU utility will automatically generate keys starting with 1 and incrementing for each item.

```
MenuStatusType =          {Status of a menu operation}
(MenuSuccess,             {Operation successful}
 MenuNoMem,               {Insufficient memory}
 MenuFileNotFound,        {Menu library file not found}
 MenuNotLibraryFile,      {File is not a library file}
 MenuIdNotFound,          {Specified library element not found}
 MenuFileReadError,       {Error while reading menu file}
 MenuFileWriteError,      {Error while writing menu file}
 MenuFileCreationError,   {Unable to create library file}
 MenuFileCorrupt,         {Menu file is corrupt}
 MenuLibraryFull          {No room in library index to add a new entry}
 );
```

MenuStatusType defines the various values returned by the MenuStatus function. Any value other than MenuSuccess indicates that an error has occurred.

```
MKType = MKSNone..MKSUser3;
```

MKType defines the class of actions performed by the menu system. It includes such actions as cursor up and down, item select, and menu exit. See the constants above for further details.

```
Orientation = (Vertical, Horizontal, NoOrient);
```

Each submenu in a menu system may have either of two orientations: vertical or horizontal. The NoOrient value is used internally to trap errors and should not be specified by an application.

Variables

```
MenuGetKeyPtr : Pointer;
```

Points to a routine that will return the next keystroke. See Appendix B for details.

```
MenuHelpPtr : Pointer;
```

Points to a routine to be called when the help command is given. See Appendix B for details.

```
MenuCmdNum : MKType;
```

The last menu command given to MenuChoice. Can be used to determine whether the user exited with a normal selection (<Enter> was pressed), alpha selection, or hot key (MenuCmdNum = MKSSelect), with a user-defined exit command (MenuCmdNum = MKSUser0..MKSUser3), or by pressing <Esc> at the top level of a menu (MenuCmdNum = MKSExit). Since the keystrokes associated with each menu command are installable, it is often preferable to evaluate MenuCmdNum after calling MenuChoice rather than to look at the SelectKey parameter MenuChoice also returns.

Declaration

```
procedure CheckMenuStatus(Mstatus : MenuStatusType);
```

Purpose

Check menu status, report and halt on any error.

Comments

CheckMenuStatus offers a simple (and somewhat brutal) error reporting mechanism for applications using TPMENU. If the menu status passed in as a parameter indicates an error, CheckMenuStatus writes a message and halts the program.

This procedure is provided for convenience during development of applications using TPMENU. We do not recommend that it be used by production code. See the MenuStatus function for a more delicate means of probing for errors.

Example

```
CheckMenuStatus(MenuStatus);
```

Passes the current menu status to CheckMenuStatus. If no error is pending, CheckMenuStatus just returns to the caller. Otherwise, it writes an error message and halts.

See Also

MenuStatus

DisableMenuItem

Declaration

```
procedure DisableMenuItem(Mnu : Menu; MKey : MenuKey);
```

Purpose

Prevent the user from selecting the specified item in the specified menu.

Comments

It is sometimes useful for a menu system to be context-sensitive. This means that under some conditions, a user of the menu system cannot select certain menu items. The MAKEMENU utility provides an example: when no menu system has been entered, it doesn't make sense to be able to generate menu source code, since there isn't anything to generate. MAKEMENU therefore disables this menu item (and others) under this condition, and reenables it later.

When you disable a menu item using DisableMenuItem, TPMENU does two things: it draws that item in a different color (specified using the DisabledColor element of the color array); and it prevents any cursor movements from selecting the item. Thus the item remains visible, but looks different and cannot be selected. Note that DisableMenuItem does not immediately redraw the menu that the item belongs to, so in most cases, you will want to call RedrawMenu after disabling a menu item.

To call DisableMenuItem, you must know the MenuKey of the item. This key is a unique scalar value that you specify when the menu item is defined. You'll find it as the fourth parameter to MenuItem or MenuItemPtr.

You can reenable the item at any time by calling EnableMenuItem. It doesn't hurt to disable an item more than once.

If you've assigned a hot key to an item (as does MAKEMENU using control keys), it is your responsibility to disable it by calling SetMenuSelectKeys after modifying the set of hot key characters. DisableMenuItem cannot disable the hot key for you.

Examples

```
const
  HotKeySet : MenuCharSet = [^R, ^W];
  ReadFileKey = 1;
  WriteFileKey = 2;
var
  Mnu : Menu;
begin
  {Define menu}
  Mnu := NewMenu(HotKeySet, nil);
  ...
  MenuItem('Read file', 1, 1, ReadFileKey, '');
  MenuItem('Write file', 2, 1, WriteFileKey, '');
  ...
```

```
    if not OutputReady then begin
      {Disable menu selection for writing file}
      DisableMenuItem(Mnu, WriteFileKey);
      {Disable hot key for writing file}
      HotKeySet := HotKeySet-[^W];
      SetMenuSelectKeys(Mnu, HotKeySet);
      {Update the screen to show the change}
      RedrawMenu(Mnu);
    end;
    ...
    if OutputReady then begin
      {Enable menu selection for writing file}
      EnableMenuItem(Mnu, WriteFileKey);
      {Enable hot key for writing file}
      HotKeySet := HotKeySet+[^W];
      SetMenuSelectKeys(Mnu, HotKeySet);
      {Update the screen to show the change}
      RedrawMenu(Mnu);
    end;
  end.
```

This example defines a hypothetical menu system with selections for reading and writing files. When no output is ready to be written, we disable the menu item that causes a file to be written. We also disable the hot key associated with writing a file. Later, when output is ready, we reenable the selection.

MAKEMENU puts these capabilities to good use. See MAKEMENU.INC for a more thorough example.

See Also

EnableMenuItem
SetMenuSelectKeys
RedrawMenu

DisableMenuMouse / DisposeMenu

Declaration

```
procedure DisableMenuMouse;
```

Purpose

Disable mouse control of menus.

Comments

This routine undoes the actions of EnableMenuMouse. For many programs, you'll never need to call DisableMenuMouse, since its essential functions are handled automatically by a Turbo Pascal exit routine. In some cases, particularly in memory resident programs, you may need to call DisableMenuMouse in order to disable the mouse functions temporarily.

Note that DisableMenuMouse does not hide the mouse cursor. You can call HideMouse to do that if appropriate.

Examples

```
{TSR pops up}
...
EnableMenuMouse;
repeat
  Key := MenuChoice(Main, Ch);
until (MenuCmdNum = MKSSelect) and (Key = 6);
EraseMenu(Main, False);
DisableMenuMouse;
...
{TSR returns}
```

See Also

EnableMenuMouse

Declaration

```
procedure DisposeMenu(Mnu : Menu);
```

Purpose

Dispose of all menu heap space.

Comments

TPMENU uses heap space for virtually all of its data structures. When a new menu system is initialized, space is allocated using the GetMem procedure. If an application finishes with a particular menu system, it should call DisposeMenu which uses FreeMem to reclaim all heap space.

Example

```
DisposeMenu(M)
```

Frees heap space for menu M.

See Also

MenuItem

NewMenu

DrawMenuFromStack / EnableMenuItem

Declaration

```
procedure DrawMenuFromStack(Mnu : Menu; var TStack : MenuStackP);
```

Purpose

Draw a menu system using previously saved stack of items.

Comments

In a menu-driven application, situations may arise where you wish to erase the current set of menus, update the underlying screen, and then restore the menus to their prior state. Together with EraseMenuOntoStack, DrawMenuFromStack provides the means to this end.

The second parameter to DrawMenuFromStack is a pointer to a stack initialized previously by EraseMenuOntoStack.

Example

```
var M : Menu;
    MStack : MenuStackP;
begin
  {Erase current menus}
  EraseMenuOntoStack(M, MStack);
  {Screen is restored, perform screen updates here}
  ...
  {Redraw selected menus}
  DrawMenuFromStack(M, MStack);
end;
```

See Also

EraseMenuOntoStack

Declaration

```
procedure EnableMenuItem(Mnu : Menu; MKey : MenuKey);
```

Purpose

Allow the user to select the specified menu item in the specified menu.

Comments

Call this procedure to reenable a menu item previously disabled by calling DisableMenuItem. See the entry for that routine for details.

It doesn't hurt to enable a menu item more than once.

Examples

See the example for DisableMenuItem.

See Also

DisableMenuItem

SetMenuSelectKeys

EnableMenuMouse

Declaration

```
procedure EnableMenuMouse;
```

Purpose

Enable mouse control of menus.

Comments

Call this routine to activate mouse control of menu systems. The routine checks to assure that a mouse is installed, then sets an internal flag so that other menu routines will utilize the mouse. You should call EnableMenuMouse once prior to calling TPMENU's MenuChoice routine.

Since TPMENU's mouse control depends on the TPMOUSE keyboard event handler, EnableMenuMouse also calls TPMOUSE's EnableEventHandling routine. There is no harm done if you've called this routine previously. Be aware, though, that any mouse event handler which you've previously activated will lose control after the call to EnableMenuMouse.

EnableMenuMouse does not modify the current mouse window, the mouse cursor type, or the mickey-to-pixel ratio, nor does it display the mouse cursor. Your program should set those values as needed by calling the appropriate TPMOUSE routines. You may use either a hardware or software mouse cursor.

When the mouse is enabled, TPMENU performs the following actions during menu choices. When the left mouse button is activated, TPMENU checks to see whether the mouse cursor is pointing to an item on the current submenu. If so, that item is selected just as if <Enter> were pressed. Failing that, TPMENU checks to see whether the mouse points to an item in the parent menu of the current submenu. If so, and if the parent menu is not already active, TPMENU selects the parent item. The right mouse button is treated just like the <Esc> key. When both mouse buttons are activated, TPMENU acts just as if <F1>, the help key, were pressed.

The actions performed when mouse buttons are activated may be configured in the same way that keystrokes are installed. See Appendix B for details.

Note that mouse actions occur when the buttons are *released*, not when they are first pressed. This technique allows reliable use of commands that require both buttons to be pressed simultaneously. Change this behavior by toggling the WaitForButtonRelease constant in TPMOUSE. After that change, mouse actions occur immediately after a mouse button is depressed.

TPMENU detects that the mouse cursor is "on" a particular item by comparing the cursor to the same region that is highlighted when the item is selected.

In a memory resident program using TPMENU, you should not enable the mouse prior to going resident. Instead, you should enable the mouse after popping up, and disable it again before returning. There are additional considerations when writing a memory resident program that uses the mouse. See Chapter 2, Section G for details.

Examples

```
program MenuDemo;
uses
  TpCrt,
  TpMouse,
  TpMenu;
var
  Ch : Char;
  Main : Menu;
  Key : MenuKey;

 procedure InitMenu(var M : Menu);
 begin
   ...
 end;

begin
  InitMenu(Main);
  EnableMenuMouse;
  repeat
    Key := MenuChoice(Main, Ch);
    GoToXy(1, 25);
    Write('Selection=', Key, ' with keystroke code of ', Ord(Ch));
    Delay(1000);
    EraseMenu(Main, False);
  until (MenuCmdNum = MKSSelect) and (Key = 6);
end.
```

Shows how easy it is to enable mouse support for the menu system. InitMenu is the same routine shown in the example at the beginning of this section. Prior to the call to EnableMenuMouse, you may wish to insert additional calls to routines from TPMOUSE, like SoftMouseCursor, HardMouseCursor, or MouseWindow.

The MAKEMENU utility makes full use of the mouse capabilities of TPMENU. See that program for a real-world example.

See Also

DisableMenuMouse

EnableEventHandling (TPMOUSE)

EraseCurrentSubMenu

Declaration

```
procedure EraseCurrentSubMenu(Mnu : Menu);
```

Purpose

Erase the currently displayed submenu.

Comments

After a user makes a selection from within MenuChoice, the menu system remains visible on-screen. Your program can then update the screen in any of several ways:

- leave the screen unchanged, and later return to the same menu item by calling MenuChoice again
- erase the entire menu system with EraseMenu, and later enter the menu system at the top level with MenuChoice
- erase the entire menu system with EraseMenu, and later enter the menu system at any specified item by calling SelectMenuItem prior to calling MenuChoice
- erase the current submenu with EraseCurrentSubMenu, and later enter the menu system at the parent of the previously selected item with MenuChoice

Hence, you call EraseCurrentSubMenu when you want to erase the most recently selected submenu from the screen. If you know that several nested submenus are visible, you can call EraseCurrentSubMenu repeatedly.

Examples

```
repeat
  MKey := MenuChoice(Mnu, SelectChar);
  if MenuCmdNum = MKSSelect then
    case MKey of
      1 : {Want to erase current submenu when this item is selected}
        begin
          {Perform whatever actions go with this item}
          {Erase the submenu before getting the next menu choice}
          EraseCurrentSubMenu(Mnu);
        end;
      2 : {Perform whatever actions go with this item} ;
    end;
until Done;
```

For this menu system, item 1 performs an action that we don't want to immediately repeat. Therefore, we erase its submenu before looping to get the next menu choice.

```
with Mnu^ do
  while Visible and (Active <> Root) do
    EraseCurrentSubMenu(Mnu);
```

This loop erases all submenus of menu Mnu until only the root menu remains visible.

See Also

EraseMenu

Declaration

```
procedure EraseMenu(Mnu : Menu; ResetSelections : Boolean);
```

Purpose

Erase active menus from the screen, reset selections to base if desired.

Comments

EraseMenu erases any menus currently displayed and restores the underlying screen contents. If ResetSelections is True, the current selection of all submenus is reset to the first item in each one. Otherwise, all selections are left with their current values.

After using EraseMenu, the next time an application calls MenuChoice it will display the top level submenu. If you want MenuChoice to return to a nested menu location, you should use the procedure EraseMenuOntoStack instead of EraseMenu. EraseMenuOntoStack stores information about the currently selected submenus while it erases them. DrawMenuFromStack will then restore the menu system to its prior state.

Example

```
EraseMenu(M, true);
```

Removes the menus from the screen and resets all selections to the initial one.

See Also

EraseCurrentSubMenu
EraseMenuOntoStack
MenuChoice

EraseMenuOntoStack

Declaration

```
procedure EraseMenuOntoStack(Mnu : Menu; var TStack : MenuStackP);
```

Purpose

Erase a menu system, saving the path of current selection on a stack.

Comments

In a menu-driven application, situations may arise where you wish to erase the current set of menus, update the underlying screen, and then restore the menus to their prior state. TPMENU's two procedures EraseMenuOntoStack and DrawMenuFromStack provide the means to this end.

The second parameter to EraseMenuOntoStack is a pointer which will contain the top of a stack of selected submenus. EraseMenuOntoStack removes the submenus from the screen, restores the underlying screen and builds a stack which DrawMenuFromStack can use to redraw the menu system.

Example

```
var
  M : Menu;
  MStack : MenuStackP;
begin
  {Erase current menus}
  EraseMenuOntoStack(M, MStack);
  {Screen is restored, perform screen updates here}
  ...
  {Redraw selected menus}
  DrawMenuFromStack(M, MStack);
end;
```

See the demonstration program MAKEMENU for a comprehensive example of using EraseMenuOntoStack.

See Also

DrawMenuFromStack

FindMenuItem / GetCurrentMenuKey

Declaration

```
function FindMenuItem(Mnu : Menu; MKey : MenuKey; var Item : ItemP) : Boolean;
```

Purpose

Search for item key MKey, returning True and pointer to item if found.

Comments

This function is primarily for internal use. Given an item key, it returns a pointer to the record which describes the details of that item. We use this routine to enable and disable items, and to change the name associated with items. You may find additional uses for it. Study the ItemRec declaration in TPMENU.IN0 to see all of the information contained in an item record.

Examples

```
if FindMenuItem(Mnu, MKey, Item) then
  {Key MKey was found}
  with Item^ do begin
    WriteLn('Item name is: ', string(Name^));
    WriteLn('Item is enabled: ', Enabled);
  end;
```

Finds the menu item associated with MKey and then writes out the name of the item and whether it is currently enabled.

See Also

DisableMenuItem
GetMenuItemName
SetMenuItemName

Declaration

```
function GetCurrentMenuKey(Mnu : Menu) : MenuKey;
```

Purpose

Return the key of the currently selected item.

Comments

If the menu is inactive or not visible, GetCurrentMenuKey returns MaxLongInt. Otherwise it returns the key value for the currently selected item.

See Also

MenuChoice

GetMenuItemName

Declaration

```
function GetMenuItemName(Mnu : Menu;  MKey : MenuKey;
                         var Name : string) : Boolean;
```

Purpose

Return the string naming the specified menu item. The function value is True if the item exists and False if it does not.

Comments

This routine first searches the menu system to find a key with value MKey. If the item isn't found, GetMenuItemName returns False and an empty string. Otherwise it returns the name associated with the item.

This function may be useful for building custom error messages based on menu selections.

Examples

```
const
  ReadFileKey = 1;
  WriteFileKey = 2;
var
  Mnu : Menu;
  Name : string;
begin
  {Define menu}
  Mnu := NewMenu([], nil);
  ...
  {And some items on the menu}
  MenuItem('Read file', 1, 1, ReadFileKey, '');
  MenuItem('Write file', 2, 1, WriteFileKey, '');
  ...
  {Now return the name of an item}
  if GetMenuItemName(Mnu, ReadFileKey, Name) then
    WriteLn('Name of item ReadFileKey is ', Name)
  else
    WriteLn('No item ReadFileKey found');
end.
```

The example gets the name of the item associated with MenuKey ReadFileKey, which is 'Read file'.

See Also

SetMenuItemName

Declaration

```
function MenuChoice(Mnu : Menu; var SelectKey : Char) : MenuKey;
```

Purpose

Display menu system, and allow user to make a selection.

Comments

MenuChoice is the central routine of TPMENU. An application will call other routines to build or dispose of menu systems only a few times during the course of a run. By contrast, it will call MenuChoice repeatedly whenever a menu selection is required from the user.

The first parameter to MenuChoice is a Menu previously initialized by a NewMenu or ReadMenuLib statement. MenuChoice determines whether the specified menu system is already on-screen: if not, it draws the top level submenu; otherwise it picks up where a previous call to MenuChoice left off.

Within MenuChoice, the user can use a predefined set of keys to navigate around the menu system. The following keystrokes are the default values for TPMENU, but may be modified using the techniques described in Appendix B. TPMENU supports the WordStar keystrokes as well.

Cursor keys
Change the currently selected item left, right, up, or down. They also change the currently selected submenu when it is logical to do so.

<Enter>
If the current item has no nested submenus, MenuChoice will exit, returning the MenuKey of that item as well as the keystroke used to select it (<Enter> in this case). If a submenu is attached, <Enter> selects the submenu.

<Esc>
If the current submenu is at the top level, MenuChoice will exit, returning the MenuKey of the currently selected item as well as the keystroke used to select it (<Esc> in this case). If the current submenu is nested, MenuChoice will erase it from the screen and return to the previously selected submenu.

Selection characters
If you desire, MenuChoice allows selection of menu items when an identifying character is pressed. Generally the character will be the first letter of the item name. When a matching key is pressed (case not being important), MenuChoice returns the MenuKey of the selected item and returns the <Enter> key as the selection character.

<F1>
If the global variable MenuHelpPtr is not nil, MenuChoice will call your help routine when this key is pressed. See Appendix B for details.

MenuChoice

An application may define an additional set of keystrokes for selection. For example, if a user pressed the <F10> key, MenuChoice could exit and return the MenuKey of the selected item as well as the <F10> keystroke. The application might then pop up a subsidiary window, erase it, and then call MenuChoice again to let the user complete the selection process. The additional keystroke set is defined at the time of a call to NewMenu, and may be changed at any time by calling SetMenuSelectKeys.

When MenuChoice exits, it leaves any selected menus on-screen. The menus can be erased, and the underlying screen restored, by a call to EraseMenu.

As mentioned, MenuChoice returns two items of information. In the function result is the MenuKey. A MenuKey is nominally of type LongInt. In practice, the MenuKey may hold any data type whose size is four bytes or less. The MenuKey is a unique identifier for each item within a menu system. It might be a simple integer value to use in a case statement, an enumerated type, or a pointer to a more complex structure holding further details. A MenuKey is assigned to each menu item during a call to MenuItem when the menu is initialized.

In the VAR parameter SelectKey, MenuChoice returns the keystroke used to perform the selection. Since sets are restricted to byte sized elements, TPMENU uses a simple encoding method to pack extended keystrokes into a single byte. Normal ASCII characters in the range #0..#127 are represented as usual. Extended keystrokes such as <F1> (#0#59) are represented by dropping the null character and setting the top bit of the scan code. Thus, the encoding for <F1> is #187 (#59 with its top bit set). This encoding prohibits the use of normal high bit (line drawing or foreign) characters for selection, but these are an unlikely requirement.

The TPMENU code for the command used to exit from the menu system is returned in the global variable MenuCmdNum.

Example

```
repeat
  Key := MenuChoice(Main, Ch);
  EraseMenu(Main, False);
  if MenuCmdNum = MKSSelect then
    case integer(Key) of
      1: {Perform action for item 1};
      2: {Perform action for item 2};
      ...
    end;
until (MenuCmdNum = MKSExit) or ((MenuCmdNum = MKSSelect) and (Key = ExitKey));
```

This example loops continuously until a particular menu selection is made or the <Esc> key is pressed at the top level. The case statement selects among MenuKey values assigned during menu initialization.

See Also

SetMenuSelectKeys

Declaration

```
procedure MenuHeight(Height : Byte);
```

Purpose

Set height of submenu currently being built.

Comments

TPMENU automatically adjusts submenus to the minimum size that will surround the various items in the menu. MenuHeight, along with its companion MenuWidth, allows you to select a larger menu size in cases where you may need a set of Lotus-style menus all the same width, a pulldown menu system that spans the entire screen, a menu with extra space at the bottom for a separate pop up window, or a larger menu just because it looks better that way.

You call MenuHeight while a particular submenu is being built. The best place to insert a call to MenuHeight is immediately following the SubMenu call that creates the submenu. (Source code generated by MAKEMENU always places calls to MenuHeight and MenuWidth this way.) If you wish, you may call MenuHeight at any time after the call to SubMenu and before the next call to PopSubLevel.

The Height parameter specifies the total number of rows in the menu, including the frame, if any. If the Height is less than the minimum number of rows to display all of the items, it will be ignored and the larger height used.

Examples

```
M := NewMenu([], nil);
SubMenu(1,2,1,Horizontal,Frame1,Color1,'TPMENU');
  MenuWidth(30);
  MenuItem('First',2,1,1,'');
  SubMenu(2,4,1,Vertical,Frame2,Color1,'');
    MenuWidth(20);
    MenuHeight(6);
    MenuMode(True, True, False);
    MenuItem('Alpha',1,1,2,'');
    MenuItem('Beta',2,1,3,'');
    PopSublevel;
  MenuItem('Second',12,1,4,'');
  SubMenu(11,4,1,Vertical,Frame2,Color1,'');
    MenuWidth(20);
    MenuHeight(6);
    MenuMode(True, True, False);
    MenuItem('Gamma',1,1,5,'');
    MenuItem('Delta',2,1,6,'');
    PopSublevel;
  PopSublevel;
ResetMenu(M);
```

This repeats the example at the start of this section, but adds calls to MenuHeight and MenuWidth. The top-level horizontal menu gets a width of 30 columns, larger than the default width of 17. The vertical submenus each end up 20 columns wide and 6 rows tall. These submenus also use exploding frames with shadows.

See Also

MenuMode

MenuWidth

MenuItem

Declaration

```
procedure MenuItem(NameStr : string; DisplayPosP : Byte;
                   SelectPosP : Byte; KeyP : MenuKey; HelpStr : string);
```

Purpose

Add an item to a submenu.

Comments

Call this procedure while building a new menu system. After first defining a submenu (by calling SubMenu), call MenuItem once for each item in the submenu.

Each call to MenuItem defines the name of the item (NameStr), the offset of the item from the top left corner of the submenu (DisplayPosP), the offset within the name of the selection character (SelectPosP), the MenuKey returned when the item has been selected (KeyP), and an optional help string to display while the cursor is over the item (HelpStr).

The interpretation of DisplayPosP changes depending on whether the current submenu has horizontal or vertical orientation. If horizontal, DisplayPosP means the column offset from the left edge of the submenu. If vertical, DisplayPosP means the row offset from the top edge of the submenu.

If SelectPosP is 1, the first character of the item name will display in the HiliteColor attribute. That character will also act to select the menu item. If SelectPosP is set to 0, no character will be highlighted or used in selection.

KeyP is nominally of type MenuKey, but actually it may be any data type of four bytes or less. You can typecast other types to MenuKey when calling MenuItem. We recommend that menu keys have integer values in the range from 1 to 10000 in order to provide compatibility with the context-sensitive help system TPHELP.

MenuItem allocates heap space to store NameStr and HelpStr. If these strings are already stored as static data, you should call MenuItemPtr rather than MenuItem. MenuItemPtr does not allocate additional heap space for the strings. MenuItem may return an error if insufficient space is available to store the strings.

MenuItem inserts the new item into the currently active submenu immediately after the currently active item and makes the newly added item the current item.

By using the MAKEMENU utility to design menu systems interactively, you can avoid worrying about many of MenuItem's details.

Example

```
SubMenu(1,2,25,Horizontal,Frame1,Color1,'TEST');
MenuItem('File', 2, 1, MenuKey(FileOps), 'Help for file operations');
```

Adds the first menu item to the horizontal submenu entitled 'TEST'. The new item is named 'File', its position is two columns right of the submenu's edge, the letter 'F' is a selection character, it will return a MenuKey whose value is FileOps, and it will display a brief help line on row 25 whenever it is selected.

See Also

MenuItemPtr

SubMenu

Declaration

```
procedure MenuItemPtr(NamePtr : Pointer; DisplayPosP : Byte; SelectPosP : Byte;
                      KeyP : MenuKey; HelpPtr : Pointer);
```

Purpose

Add an item to a submenu.

Comments

MenuItemPtr works just like MenuItem, with one exception. MenuItemPtr does not allocate heap space to store the Name and Help strings for the item. Instead the calling application provides pointers to these strings, which must already be located in some static storage location.

MenuItemPtr is useful if the strings used by a menu are defined as typed constants, or if they are read in from a file and stored in global variables or on the heap. Do *not* use MenuItemPtr if the menu strings are stored in local variables.

Generally, MenuItemPtr is more memory efficient than MenuItem. Literal strings passed to MenuItem are stored once in the code segment by the compiler, and another time on the heap by MenuItem. By contrast, typed constant strings used with MenuItemPtr are stored a single time in the data segment. Using MenuItemPtr to avoid this doubling effect may accrue significant savings for large menu systems.

See MenuItem for further details about other facets of MenuItemPtr.

Example

```
const
  name1 : string[4] = 'File';
  help1 : string[24] = 'Help for file operations';
begin
  ...
  MenuItemPtr(@name1, 2, 1, MenuKey(FileOps), @help1);
  ...
end;
```

Turbo's '@' operator passes a pointer to the typed constant strings on to MenuItemPtr.

See Also

MenuItem

MenuMode

Declaration

```
procedure MenuMode(Explode, Shadow, Noise : Boolean);
```

Purpose

Set mode of submenu currently being built.

Comments

TPMENU depends on the TPWINDOW unit to draw the basic outlines of each submenu. As a result, TPMENU can offer the same window modes that TPWINDOW does for special effects--exploding frames, shadowed windows, even sound effects during the explosions. By default, TPMENU uses none of these modes, and produces plain submenus without shadows, explosions, or noise.

MenuMode is used to change the default display mode of any submenu. It must be called while the submenu is being built, and should immediately follow the SubMenu call that defines the submenu. As such, each submenu of a menu system can have different display modes--some can be shadowed, others exploding, others plain, according to your tastes.

The time delay between each stage of a window explosion is a global property of the menu system. Use SetMenuDelay to change this value.

MenuMode selects the shadow mode (transparent or opaque) for shadowed menus based on the global typed constant TPWINDOW.ShadowMode.

Examples

See the example code for procedure MenuHeight.

See Also

MenuHeight
MenuWidth
SetMenuMode
SetMenuDelay

Declaration

```
function MenuStatus: MenuStatusType;
```

Purpose

Return status of previous operation.

Comments

MenuStatus works in a manner similar to Turbo's IoResult function. MenuStatus returns the current error result, and then it resets the internal error variable. See the declarations section, MenuStatusType, for the various error categories.

Example

```
MStat := MenuStatus;
if MStat <> MenuSuccess then begin
  writeln('Error in menu operation: ',ord(MStat));
  halt(1);
end;
```

See Also

CheckMenuStatus Declarations

Declaration

```
procedure MenuWidth(Width : Byte);
```

Purpose

Set width of submenu currently being built.

Comments

TPMENU automatically adjusts submenus to the minimum size that will surround the various items in the menu. MenuWidth, along with its companion MenuHeight, allows you to select a larger menu size in cases where you may need a set of Lotus-style menus all the same width, a pulldown menu system that spans the entire screen, a menu with extra space at the bottom for a separate pop up window, or a larger menu just because it looks better that way.

You call MenuWidth while a particular submenu is being built. The best place to insert a call to MenuWidth is immediately following the SubMenu call that creates the submenu. (Source code generated by MAKEMENU always places calls to MenuHeight and MenuWidth this way.) If you wish, you may call MenuWidth at any time after the call to SubMenu and before the next call to PopSubLevel.

Width specifies the total number of columns in the menu, including the frame, if any. If Width is less than the minimum number of columns to display all of the items, it will be ignored and the larger width used.

Examples

See the example code for procedure MenuHeight.

NewMenu

Declaration

```
function NewMenu(SelectKeys : MenuCharSet; UserFunc : Pointer) : Menu;
```

Purpose

Initialize a new menu system.

Comments

Call NewMenu first when initializing a new menu system. NewMenu allocates a small control block on the heap and returns a pointer to this block. The returned Menu pointer is used as the identifier for all subsequent menu operations.

NewMenu's parameters define two global properties of the menu system. SelectKeys is a set of characters that offer alternate ways to select menu items. If SelectKeys is the empty set, only the <Enter> key will allow the user to select a particular menu choice. By adding characters to SelectKeys, you might enable the user to select a menu choice by pressing <F1>, or you might provide "hot key" access to faraway portions of the menu system.

Since sets are restricted to byte-sized elements, TPMENU uses a simple encoding method to pack extended keystrokes into each element of the SelectKeys set. Normal ASCII characters in the range #0..#127 are represented as usual. Extended keystrokes such as <F1> (#0#59) are represented by dropping the null character and setting the top bit of the scan code. Thus, the encoding for <F1> is #187 (#59 with its top bit set). This encoding prohibits the use of normal high bit (line drawing or foreign) characters for selection, but these are an unlikely requirement.

The UserFunc parameter enables another powerful feature of the TPMENU system. UserFunc is a pointer to a function provided by your application. This function will be called for every menu item displayed on-screen. The function's responsibility is to customize the string displayed for the item, perhaps to add some status information. For example, consider a menu item used to toggle between insert and overwrite mode in a word processor. It is desirable for the menu item itself to display the current state of this toggle. This may be achieved by means of the UserFunc function.

If UserFunc is nil, no user function need be defined, and this feature of TPMENU will be inactive. Otherwise, UserFunc must contain the address of a function with the following declaration:

```
{$F+}
function DisplayFunc(S : string; K : MenuKey) : string;
  {-Customize the display string for each menu item}
begin
end;
{$F-}
```

Note first that the function must be compiled under the FAR model. Failing to precede it with the {$F+} directive will lead directly to a program crash. The function must take two parameters as shown, and return a string result. The function's name is arbitrary.

On entry to the user function, the string S will hold a non-customized version of

the string about to be displayed for a particular menu item. The string will have been padded to its final width, and the item name will be left-justified within the string. The parameter K will hold the unique MenuKey which identifies each item in the menu system. The user function should perform a selection based on this key and customize the string S as desired. Before returning, the function must assign the customized string to the function result. If no customization is desired for a particular menu key, the string S should be directly assigned to the function result. The string result should always be of the same length as the input string, or the screen will most likely become a mess.

See the DisplayFunc function in MAKEMENU.PAS for a detailed example of a user-defined display function.

Examples

```
Mnu := NewMenu([], nil);
```

This is the simplest way to call NewMenu. No additional selection keys are defined, and no display function is activated. The menu system will be denoted by the variable Mnu.

```
Mnu := NewMenu([#187, ^A], @DisplayFunc);
```

The <F1> and <CtrlA> keys are activated as alternate select keys, and DisplayFunc becomes the display function.

See Also

SetMenuSelectKeys

PackMenuLib

Declaration

```
procedure PackMenuLib(Iname, Oname : string);
```

Purpose

Remove obsolete menu entries from library Iname, creating Oname.

Comments

This is a utility procedure supporting TPMENU's menu libraries. When a new version of an existing menu is written to a menu library, the previous version remains in the library, but is deactivated by changing its library directory entry. If this process is repeated, the menu library will soon be wasting disk space, and may run out of directory entries, which are not automatically recycled.

The PackMenuLib procedure reads an existing menu library (Iname) and writes a cleaned up version of that library to a new file (Oname). In the process it strips out unused menu systems and clears out disabled directory entries. The output file may not be the same file as the input file, or a corrupted library will result. PackMenuLib does not enforce this requirement.

PackMenuLib will return an error if the input file is not found, if the library has no obsolete entries, or if a file I/O error occurs. PackMenuLib will overwrite the output file without warning. It is the application's responsibility to check for overwrite.

The demonstration program MAKEMENU packages PackMenuLib for easy use.

See WriteMenuLib for further information on the menu library format.

Example

```
PackMenuLib('MENU1.LIB', 'MENU2.LIB');
```

Packs MENU1.LIB, creating a new library MENU2.LIB.

See Also

MenuStatus
ReadMenuLib
WriteMenuLib

Declaration

```
procedure PopSubLevel;
```

Purpose

Pop active menu from top of menu stack.

Comments

Call PopSubLevel while initializing a multilevel menu system. PopSubLevel tells TPMENU that the current submenu is complete, and that initialization should continue with the next item in the parent menu.

Note that menu initialization is a depth-first process. If a menu item has a submenu, the submenu must be defined in full before proceeding with the next menu item.

By using the supplied program MAKEMENU, you can avoid worrying about many of the details of menu definition.

Examples

See the example in the introduction to this section for the use of PopSubLevel in a simple two-level menu system.

See Also

SubMenu

ReadMenuLib

Declaration

```
function ReadMenuLib(Fname : string; ID : string; UserFunc : Pointer) : Menu;
```

Purpose

Read a menu system from a binary menu library.

Comments

TPMENU defines a library format for menus. Each library may hold up to ten menu systems. To access a particular menu system, you must know the name of the library file and also the menu ID. The ID is a 12 character string used to differentiate between the various menus within any one library file. For more information about building a menu library, see the procedure WriteMenuLib.

ReadMenuLib takes the library file name and menu ID as parameters. It also takes as parameter a pointer to a user-defined function that customizes the appearance of menu items. If this feature is not desired, pass the value nil for the UserFunc parameter. See the NewMenu description for further details about UserFunc.

ReadMenuLib returns a pointer to a fully initialized menu system. It may instead return an error if the file or specified menu are not found, or if insufficient heap space is available to store the menu system in memory.

Example

```
Mnu := ReadMenuLib('MENU.LIB', 'MENU1', nil);
if MenuStatus <> MenuSuccess then begin
  WriteLn('Error while reading menu');
  halt;
end;
```

Reads the menu 'MENU1' from the library file 'MENU.LIB'. No display function is specified. An error check assures that the menu was successfully read.

See Also

NewMenu

WriteMenuLib

Declaration

```
procedure RedrawMenu(Mnu : Menu);
```

Purpose

Redraw the entire active menu system, including all active submenus.

Comments

This routine first erases the menu system, which is presumed to be visible at the time of the call. Then it redraws the menu system, pulling down any submenus that were active originally. This call is useful when you have just disabled or reenabled some menu items and want the change to become visible.

If the menu system is not visible when you call RedrawMenu, it does nothing.

Example

```
const
  WriteFileKey = 2;
var
  Mnu : Menu;
  MKey : MenuKey;
  SelectChar : Char;
begin
  {Define menu}
  Mnu := NewMenu([], nil);
  ...
  MenuItem('Write file', 2, 1, WriteFileKey, '');
  ...
  {Get a choice, leaving the menu system on-screen}
  MKey := MenuChoice(Mnu, SelectChar);
  ...
  if not OutputReady then begin
    {Disable menu selection for writing file}
    DisableMenuItem(Mnu, WriteFileKey);
    {Update the screen to show the change}
    RedrawMenu(Mnu);
  end;
end.
```

This example updates the screen after disabling a menu item.

See Also

DisableMenuItem

EnableMenuItem

ResetMenu / SelectMenuItem

Declaration

```
procedure ResetMenu(Mnu : Menu);
```

Purpose

Set all selections to first item.

Comments

After a menu system is built, its selections are set by default to the last item in each submenu. ResetMenu resets all selections to the first item in each submenu.

ResetMenu is sometimes useful after calling MenuChoice. Otherwise, TPMENU retains the user's previous selections until they are changed during further calls to MenuChoice.

Example

```
ResetMenu(Mnu);
```

Resets all selections of the menu system Mnu.

Declaration

```
procedure SelectMenuItem(Mnu : Menu; MKey : MenuKey);
```

Purpose

Find key MKey in the menu system Mnu and redraw the menus to select it.

Comments

Each item in a TPMENU menu system should have a unique selection value of type MenuKey. By using this routine, you can initialize a menu system so that a specified unique key becomes the current menu selection, regardless of where that key might reside in the menu system.

SelectMenuItem first assures that the specified menu key is part of the menu system and is enabled. If this is not so, SelectMenuItem exits without doing anything. If the menu system is currently on-screen and the desired menu item is not on the current submenu, SelectMenuItem next erases the complete menu system. Finally, SelectMenuItem selects the specified menu key as the current item and redraws the menu system to display it.

Example

```
SelectMenuItem(M, 10);
```

Selects item 10 from the menu system M, and draws the menu system to show that item selected.

Declaration

```
procedure SetMenuDelay(Mnu : Menu; ExpDelay : Word);
```

Purpose

Change explosion delay for entire menu system.

Comments

This routine changes the delay per stage for exploding windows. If no submenus in the menu system have explosions enabled, calling SetMenuDelay does nothing.

ExpDelay is expressed in milliseconds, and represents the delay between each succeeding frame of the exploding window.

Although the delay per stage is a global property of a menu system, each submenu may be individually designated as exploding or not. All explosions in a menu system may be temporarily disabled by setting an explosion delay of 0.

It doesn't matter whether the menu system is on-screen or not when SetMenuDelay is called.

Examples

```
var
  Mnu : Menu;
  SaveDelay : Word;
begin
  {Build menu system}
  ...
  {Save the existing explosion delay}
  SaveDelay := Mnu^.Root^.Draw.ExploDelay;
  {Disable explosions}
  SetMenuDelay(Mnu, 0);
  {Reenable explosions}
  SetMenuDelay(Mnu, SaveDelay);
end.
```

This example shows how to temporarily disable explosions in a menu system. The expression used to obtain SaveDelay applies to any menu system, just replace the first word with the name of your menu variable.

See Also

MenuMode

SetMenuMode

SetMenuItemName

Declaration

```
function SetMenuItemName(Mnu : Menu;
                         MKey : MenuKey;
                         NewName : string) : Boolean;
```

Purpose

Change the string describing the specified menu item to a new string.

Comments

This function allows you to dynamically rename menu items. Specify the item key MKey for the item you want to rename, and pass the new name for the item in the parameter NewName.

SetMenuItemName first searches the menu system for the key MKey. If the key isn't found, SetMenuItemName returns with a False function result.

SetMenuItemName forces the new name to be exactly the same length as the old by either truncating NewName or padding it with blanks as required.

After calling SetMenuItemName, you may wish to call RedrawMenu to make the name changes visible on-screen.

Example

```
const
  WriteFileKey = 2;
var
  Mnu : Menu;
  MKey : MenuKey;
  SelectChar : Char;
begin
  {Define menu}
  Mnu := NewMenu([], nil);
  ...
  MenuItem('Write file', 2, 1, WriteFileKey, '');
  ...
  {Get a choice, leaving the menu system on-screen}
  MKey := MenuChoice(Mnu, SelectChar);
  ...
  {Rename the write file item}
  if SetMenuItemName(Mnu, WriteFileKey, 'Create file') then
    {Update the screen to show the change}
    RedrawMenu(Mnu);
end.
```

Under some condition, this program renames the 'Write file' menu item to 'Create file' and updates the screen to show the change. There is a small problem here though--the new name is longer than the old and will be truncated to 'Create Fil'. To circumvent this, the original string 'Write file' should be right-padded with one blank.

See Also

GetMenuItemName

RedrawMenu

Declaration

```
procedure SetMenuMode(Mnu : Menu; Explode, Shadow, Noise : Boolean);
```

Purpose

Change display mode of an entire menu system.

Comments

SetMenuMode changes the manner in which all submenus of a menu system are displayed. After calling this routine, all submenus will act the same--exploding or not, shadowed or not, noisy or not. Noise will occur only during the process of exploding a submenu.

It doesn't matter whether the menu system is on-screen or not when SetMenuMode is called. If shadows are enabled when SetMenuMode is called, they won't appear until the menu system is redrawn.

SetMenuMode selects the shadow mode (transparent or opaque) for shadowed menus based on the global typed constant TPWINDOW.ShadowMode.

Examples

```
var
  Mnu : Menu;
begin
  {Build menu system}
  ...
  {Enable shadowing for the menu system}
  SetMenuMode(Mnu, False, True, False);
  {Redraw menu system so shadows are visible}
  RedrawMenu(Mnu);
end.
```

See Also

MenuMode

SetMenuDelay

SetMenuSelectKeys

Declaration

```
procedure SetMenuSelectKeys(Mnu : Menu; Skeys : MenuCharSet);
```

Purpose

Change the select key set of an existing menu system.

Comments

With this routine you can dynamically change the set of alternate menu selection keys. See the description of NewMenu for a complete discussion of alternate selection keys.

Why would you want to change the selection keys? Perhaps you might want to disable context sensitive help during certain operations. Perhaps you are storing menus in libraries (with a fixed set of selection keys), but want to allow customization of certain hot keys by the user.

Examples

```
SetMenuSelectKeys(Mnu, []);
```

Disables all alternate selection keys.

```
SetMenuSelectKeys(Mnu, [#187]);
```

Assigns <F1> as the only alternate selection key.

See Also

NewMenu

Declaration

```
procedure SubMenu(XLP, YLP, YhelpP : Byte; OrientP : Orientation;
                  FrameP : FrameArray; ColorsP : MenuColorArray;
                  HeaderStr : string);
```

Purpose

Add a submenu to a menu system.

Comments

When defining a new menu system, first call NewMenu and then call SubMenu one or more times to define the properties of each rectangular region of a menu system. Calls to MenuItem will add desired items to each submenu.

XLP and YLP set the screen position of the upper left corner of the submenu. Acceptable values for XLP range from 1 to 80, and for YLP from 1 to 25 (or 43 or 50). If the submenu has a frame, the upper left corner of the frame will be drawn at XLP,YLP. If it has no frame, then the menu text starts at the same position.

TPMENU's submenus are automatically sized to hold the items within them. Thus you need not specify the position of the lower right corner of the menu. However, you may explicitly set the width and height of a submenu by calling the MenuWidth and MenuHeight routines just after calling SubMenu.

YhelpP sets the position of an optional help row for the submenu. Each item within the menu can specify one line of help to be displayed starting at column 1 of the specified row. To disable this feature, specify the constant NoHelp here.

OrientP specifies the orientation of the submenu: horizontal or vertical.

FrameP determines what type of border surrounds the submenu. See the TPCRT declarations section for FrameArray for details about this six-character array. If you wish to use a Lotus-style menu with no frame, pass the special constant NoFrame for this parameter.

ColorsP defines the colors used to draw the submenu. TPMENU uses up to eight colors to draw each submenu. See the Declarations section for MenuColorType for more details. If the TPCRT variable MapColors is True, ColorsP will be mapped for better visibility on monochrome and black-and-white displays.

HeaderStr specifies a string to display centered on the top bar of the menu frame. If this string is empty, no title will appear. For Lotus-style menus, a title string is not allowed.

SubMenu returns an error if insufficient memory is available to add the menu structure.

Example

```
SubMenu(1,2,NoHelp,Horizontal,Frame1,Color1,'TPMENU');
```

Defines a submenu starting in column 1, row 2 of the screen. No help will be displayed. The menu will scroll horizontally. The menu frame and colors are stored in typed constants of the appropriate types, named Frame1 and Color1. Centered on the top bar of the submenu is the title 'TPMENU'.

WriteMenuLib

Declaration

```
procedure WriteMenuLib(Mnu : Menu; Fname : string; ID : string);
```

Purpose

Write a menu system to a binary menu library.

Comments

TPMENU defines a library format for menus. Each library may hold up to ten menu systems. To write a menu library, you must define a filename and an identifier for each particular menu system within the library. The menu ID is a 12 character string used to differentiate between the various menus within any one library file.

WriteMenuLib writes a menu previously defined in memory to a menu library file. The in-memory menu is specified with the parameter Mnu. The library file name is Fname, and the menu identifier is ID.

If the file named by Fname does not already exist, WriteMenuLib will create it. If the file exists but is not a TPMENU library file, WriteMenuLib will exit with an error. If the menu identifier specifies a menu already in the library, WriteMenuLib marks the existing version of the menu as obsolete and adds the new version to the library. In doing so, WriteMenuLib disables the menu's library directory entry but leaves the menu system itself intact within the library. As such, obsolete menus are potentially recoverable and continue to use disk space. The utility procedure PackMenuLib is provided to remove obsolete menus from a library.

With one exception, WriteMenuLib writes a complete description of the specified menu system. The exception is the user-defined display function. Since there is no guarantee that the address of such a function would be the same the next time a menu was read from a library, storing this information would be meaningless. Instead, the ReadMenuLib function accepts the specification of a user-defined display function when a menu is read from the library.

The menu system remains intact in memory even after being written to disk. Call DisposeMenu if you want to deallocate it.

WriteMenuLib will return an error if any disk error occurs. See the Declarations section for MenuStatusType for more information on possible errors.

Example

```
WriteMenuLib(Mnu, 'MENU.LIB', 'MENU1');
```

Writes the menu system Mnu to the file MENU.LIB with the identifier MENU1.

See Also

ReadMenuLib

E. TPEDIT: Line Editor

The central routine in the TPEDIT unit is ReadString, a sophisticated line editor with WordStar compatibility and lots of options. ReadString offers the following editing commands:

<Enter>
Accept line.

<Esc>, <CtrlBreak>
Quit without changing line.

<Left>, <CtrlS>
Cursor left one character.

<Right>, <CtrlD>
Cursor right character.

<CtrlLeft>, <CtrlA>
Cursor left one word.

<CtrlRight>, <CtrlF>
Cursor right one word.

<Home>, <CtrlQ><S>
Cursor to beginning of line.

<End>, <CtrlQ><D>
Cursor to end of line.

<Del>, <CtrlG>
Delete character at cursor.

<Bksp>, <CtrlH>, <CtrlBksp>
Delete character to left of cursor.

<CtrlEnd>, <CtrlQ><Y>
Delete to end of line.

<CtrlY>, <CtrlX>
Delete entire line.

<CtrlHome>
Delete from beginning of line.

<CtrlT>
Delete word to right of cursor.

<Ins>
Toggle insert mode on and off. Fat cursor indicates insert mode; thin cursor indicates overwrite mode.

<CtrlR>, <CtrlQ><L>
Restore original contents of line.

<CtrlP>
If you give this command, the next character entered will be inserted into the string no matter what. If it is a control character and EditMapCtrls is True, it will be displayed as an uppercase alphabetic character in the colors specified by the CtrlAttr parameter to ReadString, rather than the StringAttr. For example, <CtrlP><CtrlC> would display a 'C' in CtrlAttr. If EditMapCtrls is False, the control character will be displayed normally (in CtrlAttr). As a reminder that it is waiting for a control character, ReadString will temporarily display a block cursor; after a key is pressed, the cursor will resume its previous shape. Two points worth noting: First, if CtrlAttr = StringAttr and EditMapCtrls = True, this command will be ignored. Second, none of the read routines except ReadString take a CtrlAttr parameter, since control characters would never be considered valid input when entering numbers.

TPEDIT also provides mouse support, which can be enabled simply by calling EnableEditMouse. If mouse support is enabled, the following commands are available while using ReadString or any of the numeric reads:

<Click Left>
Same as pressing <Enter>.

<Click Right>
Same as pressing <Esc>.

<Click Both>
Same as pressing <F1>.

ReadCharacter and YesOrNo can also respond to the pressing of a mouse button. By default, the left button simulates the pressing of <Enter>, while pressing any other mouse button or combination of buttons will do the same thing as pressing

<Esc>. But you can change these button-to-key code assignments by altering the translation table, EditButtonCodes.

You can change the editing keys, and the mouse function assignments, using any of the three methods described in Appendix B. In addition, there are several editing options that you can select just by changing a global typed constant: force upper case (ForceUpper--defaults to False); start editing with cursor at beginning or end of string (CursorToEnd--defaults to end of string); automatically trim leading and trailing blanks (TrimBlanks--defaults to True); clear string if first character entered is ASCII (ClearFirstChar--defaults to True); treat screen coordinates as relative to current window (WindowRelative--defaults to False); map control characters to uppercase alphabetics (EditMapCtrls--defaults to True); inhibit display of characters entered (PasswordMode--defaults to False); and a default editing mode of insert or overtype (InsertByDefault--defaults to True).

Since the numeric string editing routines--ReadLongInt, ReadInteger, ReadWord, and ReadReal--use ReadString themselves, the editing commands available in them are identical. The only thing that distinguishes them, from a user's perspective, is that they perform error checking. If the user enters an invalid string or the value is outside the range you specify, an error message is displayed, and the user is given a chance to reenter the number.

The other two routines in TPEDIT, ReadCharacter and YesOrNo, are used for prompting for individual characters. ReadCharacter lets you restrict the set of valid characters that may be entered. YesOrNo normally restricts the set of valid entries to 'Y' and 'N', but if you pass either of these as the Default parameter, it will display the default answer in a box like this

```
Do you wish to continue? [Y]
```

and allow the user to select the default by pressing <Enter>.

Note for Owners of Turbo Professional 4.0

If you are upgrading to this version of Turbo Professional from a version prior to 4.04, a small change in the behavior of ReadString (and routines that call it, like ReadReal) may affect existing applications. The easiest way to describe this change is through an example. Suppose you specify that the maximum string length you want is 10 characters (via ReadString's MaxLen parameter). ReadString will block out 11 columns of the screen, so that you have room to enter 10 characters and still have one left over (where the cursor will sit when the string is at maximum length). In versions prior to 4.04, ReadString would block out only 10 columns, and would allow the cursor to go one character beyond the right of the editing box. This change improves the visual appearance of the display and avoids problems that otherwise

occur in column 80 of the screen. You can obtain the earlier behavior by setting the global typed constant HouseCursorAtEnd to False.

Constants

```
ClearFirstChar : Boolean = True;
```

If True, ReadString and the numeric reads clear the existing string if the first character entered is an ASCII character.

```
CursorToEnd : Boolean = True;
```

If True, ReadString and the numeric reads (ReadLongInt, ReadInteger, ReadWord, ReadReal) position the cursor at the end of the string when editing begins, rather than at the start of the string.

```
EditMapCtrls : Boolean = True;
```

If True, any control characters entered into a string are converted to uppercase alphabetics and displayed in the attribute indicated by the CtrlAttr parameter to ReadString. If False, they are displayed normally (in CtrlAttr).

```
EditSize : Byte = 0;
```

When EditSize is 0, the MaxLen parameter passed to ReadString determines both the maximum length of the string and the width of the edit field. If you set EditSize to a non-zero value prior to calling ReadString, however, that value is used to determine the width of the edit field, while the MaxLen parameter specifies the maximum length of the string. EditSize is thus the key if you need to be able to edit strings longer than will fit on the screen. For example, if you wanted to edit a 128-character string, but only had 40 columns of screen space for the edit field, you might call ReadString as follows:

```
EditSize := 40;
ReadString(Prompt, Row, Col, 128, Attr, Attr, Attr, Escaped, S);
EditSize := 0;
```

ReadString would then use only 40 columns for the edit field, and would scroll horizontally as necessary while editing.

```
ForceUpper : Boolean = False;
```

If True, ReadString and the numeric reads force characters to upper case.

```
HideCursorInReadChar : Boolean = False;
```

If True, ReadCharacter and YesOrNo hide the cursor temporarily.

```
HouseCursorAtEnd : Boolean = True;
```

If True, ReadString extends the entry box by one position, leaving space to house the cursor when the string being edited is full.

```
InsertByDefault : Boolean = True;
```

If True, ReadString and the numeric reads default to insert mode rather than overtype.

```
PasswordChar : Char = ^G;
```

Character used to represent text while in password mode. The ^G character is a dot in the middle of the character box.

```
PasswordMode : Boolean = False;
```

If True, ReadString will not display any of the characters in the string being edited. Instead, each character is represented on screen as PasswordChar (see below).

```
RSpadChar : Char = ' ';
```

This character is used to pad the string being edited when it is displayed. A common alternative to ' ' is '_'.

```
RSnone      = 00; {Not a command}
RSchar      = 01; {A character to enter the string}
RSctrlChar  = 02; {Accept control character}
RSenter     = 03; {Accept current string and quit}
RSquit      = 04; {Restore default string and quit}
RSrestore   = 05; {Restore default and continue}
RShome      = 06; {Cursor to beginning of line}
RSend       = 07; {Cursor to end of line}
RSleft      = 08; {Cursor left by one character}
RSright     = 09; {Cursor right by one character}
RSwordLeft  = 10; {Cursor left one word}
RSwordRight = 11; {Cursor right one word}
RSback      = 12; {Backspace one character}
RSdel       = 13; {Delete current character}
RSdelEol    = 14; {Delete from cursor to end of line}
RSdelBol    = 15; {Delete from beginning of line to cursor}
RSdelLine   = 16; {Delete entire line}
RSdelWord   = 17; {Delete word to right of cursor}
RSins       = 18; {Toggle insert mode}
RShelp      = 19; {Invoke help routine}
RSuser0     = 20; {user-defined exit commands}
...
RSuser9     = 29;
```

These are the codes for each of ReadString's editing commands. Most of them are self-explanatory. The last 10 (RSuser0..RSuser9) are set aside to allow user-defined "exit commands" to be implemented. If one of these commands is executed, ReadString will do the same things that it would if <Enter> were pressed; it will then be up to you to check the global variable RScommand and perform the function associated with the command just issued. For more details on how to implement user-defined commands, see Appendix B.

```
ShowReadChar : Boolean = False;
```

If True, ReadCharacter and YesOrNo display the character pressed.

```
TrimBlanks : Boolean = True;
```

If True, ReadString automatically trims leading and trailing blanks (forced on temporarily for the numeric read routines).

```
WindowRelative : Boolean = False;
```

If WindowRelative is true, all of the routines in TPEDIT will interpret the screen coordinates passed to them as being relative to the current window rather than absolute. It is up to the programmer, however, to insure that the editing field will fit within the current window.

Types

```
CharSet = set of Char;
```

Generic set of characters.

```
{$IFDEF UseMouse}
 EditButtonCodes : array[$E9..$EF] of Word = (
  $011B,  {all three buttons          = <Esc>}
  $011B,  {right and center buttons   = <Esc>}
  $011B,  {left and center buttons    = <Esc>}
  $011B,  {center button              = <Esc>}
  $3B00,  {both buttons               = <F1>}
  $011B,  {right button               = <Esc>}
  $1C0D); {left button                = <Enter>}
```

This table is used by ReadCharacter and YesOrNo in order to translate mouse buttons to key codes. As you can see, pressing the left mouse button does the same thing as pressing <Enter>, pressing both buttons does the same thing as pressing <F1>, and pressing any other button or combination of buttons does the same thing as pressing <Esc>. You are, of course, free to alter this table if you wish.

```
RStype = RSnone..RSuser9;
```

A ReadString editing command.

Variables

```
EditKeyPtr : Pointer;
```

Points to a routine that will return the next keystroke. See Appendix B for details.

```
EditHelpPtr : Pointer;
```

Points to a routine to be called when the help command is given. See Appendix B for details.

```
EditHelpIndex : Word;
```

Current help topic. This value is passed to the routine pointed to by EditHelpPtr. See Appendix B for details.

```
RSCommand : RStype;
```

The last editing command given in ReadString (or any of the numeric reads).

```
RSChWord : Word;
```

Last key entered in ReadString (or any of the numeric reads). By checking the value of this variable, you can determine whether <Enter>, <Esc>, or <CtrlBreak> was used to exit from the editor, for example.

DisableEditMouse / EnableEditMouse

Declaration

```
procedure DisableEditMouse;
```

Purpose

Disable mouse support in TPEDIT.

Comments

This routine deactivates mouse support in TPEDIT. Besides clearing the internal flag set by EnableEditMouse, it deactivates the mouse event handler in TPMOUSE and sets EditKeyPtr to point to ReadKeyWord (TPCRT).

Declaration

```
procedure EnableEditMouse;
```

Purpose

Enable mouse support in TPEDIT.

Comments

This routine activates mouse support in TPEDIT. In addition to setting an internal flag for use by other routines in TPEDIT, it activates the mouse event handler in TPMOUSE and sets EditKeyPtr to point to the ReadKeyOrButton routine in TPMOUSE. See the introduction to TPMOUSE in section 2.G for more details.

See Also

EnableEventHandling (TPMOUSE) ReadKeyOrButton (TPMOUSE)

Declaration

```
procedure ReadCharacter(Prompt : string; Row, Col : Byte; PromptAttr : Byte;
                        Accept : CharSet; var Ch : Char);
```

Purpose

Display a prompt and wait for a key in Accept.

Comments

This function offers a convenient means of prompting a user for entry of a single character while restricting the range of acceptable responses.

Prompt is a string to be displayed on the screen at Row,Col. PromptAttr is the video attribute to use when displaying Prompt. Accept is the set of acceptable characters.

The first acceptable character entered is returned in Ch. All characters are automatically converted to upper case, so "['A'..'Z']" is treated as equivalent to "['a'..'z','A'..'Z']". Extended scan codes are ignored.

If you need to use ReadCharacter to read function keys, cursor keys, etc., please refer to the discussion of the MenuChoice routine in TPMENU. ReadCharacter uses the same bit-twiddling technique employed in TPMENU for packing extended keystrokes into a single byte.

Note that the global typed constants HideCursorInReadChar and ShowReadChar allow you to customize the behavior of ReadCharacter.

Examples

```
var
  Ch : Char;
begin
  ShowReadChar := True;
  ReadCharacter('Enter a letter of the alphabet: ', 1, 1, $07, ['A'..'Z'], Ch);
  ReadCharacter('Enter a number (0..9): ', 2, 1, $07, ['0'..'9'], Ch);
  ReadCharacter('Enter any character: ', 3, 1, $07, [#0..#255], Ch);
end;
```

Using ReadCharacter to prompt for entry of an alphabetic character, a numeral, or any character.

See Also

MenuChoice (TPMENU) YesOrNo

ReadInteger

Declaration

```
procedure ReadInteger(Prompt : string; Row, Col, MaxLen : Byte;
                      PromptAttr, StringAttr : Byte; NLo, NHi : Integer;
                      var Escaped : Boolean; var N : Integer);
```

Purpose

Prompt for and receive an integer argument in the range NLo to NHi.

Comments

ReadInteger is essentially identical to ReadLongInt, except that the number entered is forced to be in the range -32768..32767 if NLo (lowest acceptable number) and NHi (highest acceptable number) are equal.

Examples

See the documentation for ReadLongInt (TPEDIT) for examples.

See Also

ReadLongInt
ReadString
ReadWord

Declaration

```
procedure ReadLongInt(Prompt : string; Row, Col, MaxLen : Byte;
                      PromptAttr, StringAttr : Byte; LLo, LHi : LongInt;
                      var Escaped : Boolean; var L : LongInt);
```

Purpose

Prompt for and receive a long integer argument in the range LLo to LHi.

Comments

This function offers a convenient means of prompting a user for entry of a string representing a long integer. ReadLongInt converts the default value (L) to a string, lets the user edit the string, then converts the string back to a long integer, returned in L. If the user presses <Esc>, L is left unchanged, and Escaped is set to True.

Prompt is a string to be displayed on the screen at Row,Col (the string to be edited is displayed to its right). MaxLen is the maximum length of the string to be entered (i.e., the width of the editing field). PromptAttr is the video attribute to use when displaying Prompt, and StringAttr is the attribute to use when displaying the string being edited.

If LLo (lowest acceptable number) and LHi (highest acceptable number) are equal, the number entered is forced to be in the range -2,147,483,648.. 2,147,483,647. Otherwise, the number is forced to be in the range LLo..LHi. If an invalid number is entered, either an error message is displayed (if MaxLen is >= the length of the error message) or a beep is produced. In either case, the user is given a chance to reenter the number.

Note that the global typed constants ForceUpper, CursorToEnd, and InsertByDefault allow you to customize the behavior of ReadLongInt, since it simply functions as a "shell" around ReadString.

Examples

```
var
  L : LongInt;
  Escaped : Boolean;
begin
  L := 0;
  ReadLongInt('Enter any long integer: ', 1, 1, 11, $07, $0F, 0, 0,
    Escaped, L);
  if Escaped then Exit;
  L := 0;
  ReadLongInt('Enter a number in the range -100..100: ', 2, 1, 4, $07, $70,
    -100, 100, Escaped, L);
  if Escaped then Exit;
end;
```

See Also

ReadString

ReadReal

Declaration

```
procedure ReadReal(Prompt : string; Row, Col, MaxLen : Byte;
                   PromptAttr, StringAttr : Byte; DecimalPlaces : Byte;
                   RLo, RHi : Float; var Escaped : Boolean; var R : Float);
```

Purpose

Prompt for and receive a real argument in the range RLo to RHi.

Comments

This function offers a convenient means of prompting a user for entry of a string representing a real number. ReadReal converts the default value (R) to a string, lets the user edit the string, then converts the string back to a real, returned in R. If the user presses <Esc>, R is left unchanged, and Escaped is set to True.

DecimalPlaces determines how R is formatted when it is initially converted to a string. For example, if R is 1.2345 and DecimalPlaces is 2, R will be displayed as '1.23'. Prompt is a string to be displayed on the screen at Row,Col (the string to be edited is displayed to its right). MaxLen is the maximum length of the string to be entered (i.e., the width of the editing field). PromptAttr is the video attribute to use when displaying Prompt, and StringAttr is the attribute to use when displaying the string being edited.

R, RLo, and RHi are all of type Float, defined in the TPSTRING unit. If TPSTRING is compiled with $N-, a Float is a six-byte Real; if it is compiled with $N+, Float is an Extended.

If RLo (the lowest acceptable number) and LHi (the highest) are equal, no range checking is performed, and any valid real may be entered. Otherwise, the number is forced to be in the range RLo..RHi. If an invalid number is entered, either an error message is displayed (if MaxLen is >= the length of the error message) or a beep is produced. In either case, the user is given a chance to reenter the number.

Note that the global typed constants InsertByDefault, CursorToEnd, and ForceUpper allow you to customize the behavior of ReadReal, since it simply functions as a "shell" around ReadString.

Examples

```
var R : Real;
    Escaped : Boolean;
...
R := 0.0;
ReadReal('Enter any number: ', 1, 1, 20, $07, $0F, 2, 0.0, 0.0, Escaped, R);
if Escaped then Exit;
R := 0.0;
ReadReal('Enter a number [0.0..100.0]: ', 2, 1, 5, $07, $70, 1, 0.0, 100.0,
  Escaped, R);
if Escaped then Exit;
```

See Also

Float (TPSTRING)

ReadString

Declaration

```
procedure ReadString(Prompt : string; Row, Col, MaxLen : Byte;
                     PromptAttr, StringAttr, CtrlAttr : Byte;
                     var Escaped : Boolean; var S : string);
```

Purpose

Prompt for and receive a string.

Comments

The central routine in TPEDIT, ReadString is a full-featured line editor offering WordStar compatibility and several useful options that allow you to control its behavior.

Prompt is a string to be displayed on the screen at Row,Col. (Row and Col are treated as absolute unless the global typed constant WindowRelative is True.) S, the string to be edited, is displayed to its right. MaxLen is the maximum length of the string; it must be in the range 1..254. PromptAttr is the video attribute to use when displaying Prompt, StringAttr is the attribute to use when displaying the string being edited, and CtrlAttr is the attribute used to display control characters. If the <Esc> key is pressed, S is returned unaltered, and Escaped is set to True.

Note that the global typed constants ForceUpper, CursorToEnd, TrimBlanks, EditMapCtrls, PasswordMode, and InsertByDefault allow you to customize the behavior of ReadString.

For a complete list of the editing commands recognized by ReadString, see the overview at the beginning of this section.

Examples

```
var
  Name, Nickname, Address : String;
  Escaped : Boolean;
begin
  Name := 'John Doe';
  ReadString('Name: ', 1, 1, 30, $07, $70, $70, Escaped, Name);
  if Escaped then Exit;
  Nickname := 'Cuddles';
  ReadString('Nickname: ', 2, 1, 30, $07, $0F, $0F, Escaped, Nickname);
  if Escaped then Exit;
  Address := '123 Main St.';
  ReadString('Address: ', 3, 1, 25, $07, $0F, $0F, Escaped, Address);
  if Escaped then Exit;
end;
```

See Also

EditString (TPENTRY)

ReadWord

Declaration

```
procedure ReadWord(Prompt : string; Row, Col, MaxLen : Byte;
                   PromptAttr, StringAttr : Byte; WLo, WHi : Word;
                   var Escaped : Boolean; var W : Word);
```

Purpose

Prompt for and receive a word argument in the range WLo to WHi.

Comments

ReadWord is essentially identical to ReadLongInt, except that the number entered is forced to be in the range 0..65535 if WLo (lowest acceptable number) and WHi (highest acceptable number) are equal.

Examples

See the documentation for ReadLongInt (TPEDIT) for examples.

See Also

ReadInteger
ReadLongInt
ReadString

Declaration

```
function YesOrNo(Prompt : string; Row, Col : Byte;
                 PromptAttr : Byte; Default : Char) : Boolean;
```

Purpose

Return True for yes, False for no.

Comments

This function offers a convenient means of prompting a user for a response to a yes-or-no question.

Prompt is a string to be displayed on the screen at Row,Col. PromptAttr is the video attribute to use when displaying Prompt. Default can be any character. Normally, YesOrNo will accept only <Y> or <N> as an acceptable response. However, if Upcase(Default) is either 'Y' or 'N', YesOrNo will treat it as the default choice, add ' [Y]' or ' [N]' to the end of Prompt, and assume that <Enter> signifies the user's acceptance of the default choice. The function returns True if <Y> is entered (or a default choice of 'Y' is accepted), and False if <N> is entered (or a default choice of 'N' is accepted).

Note that the global typed constants HideCursorInReadChar and ShowReadChar allow you to customize the behavior of YesOrNo.

Examples

```
if not YesOrNo('Do you wish to continue?', 1, 1, $07, #0) then Exit;
```

Displays 'Do you wish to continue?' and waits for <Y> or <N> to be pressed.

```
if not YesOrNo('Do you wish to continue?', 1, 1, $07, 'Y') then Exit;
```

Displays 'Do you wish to continue? [Y]' and waits for <Y>, <N>, or <Enter> to be pressed. If the user presses <Enter>, the function will return True.

```
if not YesOrNo('Do you wish to continue?', 1, 1, $07, 'N') then Exit;
```

Displays 'Do you wish to continue? [N]' and waits for <Y>, <N>, or <Enter> to be pressed. If the user presses <Enter>, the function will return False.

See Also

ReadCharacter

F. TPHELP: Popup Help

If you've ever developed a context-sensitive help system for an application, you know that it takes substantial effort to write one that is efficient and easy to use. The TPHELP unit offers a powerful, easy method to incorporate help screens into your programs. Writing the help text itself is the only major task left for you to do.

The popup help facility really consists of two parts. A unit, TPHELP.TPU, is compiled into a program that needs to offer help. In addition, a "help file compiler," MAKEHELP.EXE, will convert a text file containing the desired information to an indexed binary file for use by TPHELP itself.

The binary help file can be linked directly into the target program, so that no additional files are required by the end user of the application. Alternatively, for hefty help files or large applications, the file may be kept separate and read into memory on demand by TPHELP.

The help system supports up to 10,000 help topics in one file, each of which may have up to 20 pages of help information. In addition, each help topic may contain cross-references to other help topics, with up to 50 cross-references for each topic. Help is displayed in a pop-up window through which the user may browse. Various formatting controls allow help to appear in up to six different video attributes. The help compiler optionally word-wraps and paginates the original help text, under the control of directives that you insert.

Within an application program, help may be selected in a number of ways. It may be chosen directly by index number, by topic name, by picking information off the current screen, by popping up a master index, by cross-referencing to related items, or by referring to a stack of previously selected help items. Many of these options are invoked automatically with no further effort on your part. The TPHELP fully supports the mouse as well, leading to even friendlier help systems.

Before discussing the individual data types and procedures of the TPHELP unit, we'll describe the methodology for developing a help system.

A Simple Help Example

The following example shows one method of incorporating help in a program.

```
program HelpDemo;
uses
  Dos, TpDos, TpString, TpCrt, TpWindow, TpCmd, TpPick, TpHelp;
const
  HelpColors : HelpColorArray = ($07, $07, $0F, $70, $09, $70, $0F, $01);
var
```

```
  Help : HelpPtr;
  Status : Word;
  Topic : Word;
begin
  ClrScr;
  Status := OpenHelpFile('DEMO.HLP',   {Name of binary help file}
                         10, 2,        {Upper left corner of help window}
                         22, 2,        {Lower row, and columns in help index}
                         HelpColors,   {Colors to use}
                         Help);        {Help descriptor}
  if Status <> 0 then begin
    case Status of
        2 : WriteLn('Help file DEMO.HLP not found');
      100 : WriteLn('Unexpected end of file reading DEMO.HLP');
      106 : WriteLn('DEMO.HLP is not a valid help file');
      203 : WriteLn('Insufficient heap space available for help system');
    else
      WriteLn('Help initialization error ', Status);
    end;
    Halt;
  end;

  {Customize the help window display}
  TpWindow.Shadow := True;
  TpWindow.ShadowAttr := $70;

  repeat
    {Pick a topic from the help index}
    Topic := PickHelp(Help, 2, 2, 10, 1);
    if Topic <> 0 then
      {Display help for that topic}
      if not ShowHelp(Help, Topic) then
        WriteLn('Unable to display help for topic ', Topic);
  until Topic = 0;

  {Close help file and release memory}
  CloseHelp(Help);
end.
```

HelpColors is an array of video attributes that will be used while TPHELP displays the help information. We'll discuss what each element means later. You'll probably want to have two sets of attributes, one for when your program is run on a color monitor, and another for monochrome.

The Help variable is your program's main hook to TPHELP. This variable points to a record containing everything that TPHELP needs to know about your help system. The Help variable can be initialized with a call to either OpenHelpFile or OpenHelpMem. Our example reads the help from a file, DEMO.HLP, which has previously been generated using the utility MAKEHELP.

Note the coordinates we pass to OpenHelpFile. The first two numeric values specify the upper left corner of the help window, including its frame, if any. The next value is

the lower row for the help window, again including frame. The width of the help window is encoded into the help file itself while MAKEHELP word-wraps the text, and hence no right-hand column needs to be passed as a parameter to OpenHelpFile. The final numeric parameter specifies the number of columns of items to be used in a help index. If the user presses <F1> while help is on-screen, TPHELP automatically redraws the help window to display a list of all indexed help topics. Depending on the width and number of help topics, you may wish to have 1, 2, or more columns of topics appear side-by-side.

OpenHelpFile first searches for DEMO.HLP. (It looks in the current directory, the directory of the main program, and on the DOS PATH to find it.) Then OpenHelpFile opens the file and confirms the file has the proper format. Finally it allocates some buffer space on the heap and returns a status word. In our example, we check this status word and halt if any error occurred.

Since TPHELP builds upon the routines in TPWINDOW, we can take advantage of all the window formatting techniques available there. In this example, we activate window shadowing.

Next, we call the PickHelp routine from TPHELP, which displays a list of help topics and lets us choose one. PickHelp returns the topic number, or the value 0 if no topic was chosen.

ShowHelp then pops up a window to display the help text for that topic. While within ShowHelp, the user can page around within the topic or select another topic. When ShowHelp returns, the screen will have been restored to its previous state.

Finally, the example program calls CloseHelp, which closes the help file and releases all memory allocated for the help system.

Help File Directives

The information displayed by the help system starts out as a text file that you must write. You put directives in that text file to tell the help compiler how to format and interpret it. You then run the help compiler, MAKEHELP, to create a formatted, indexed, compressed file that can be read by TPHELP.

We typically use the extension .TXT to indicate a help text file, and reserve the extension .HLP for the binary help file produced by MAKEHELP. The help system does not require this, however. Directives in the TXT file are denoted by an exclamation point in the first column of the line, immediately followed by the directive name, followed by optional modifiers separated by one or more spaces. The following directives are accepted by the help compiler.

!WIDTH Width
Specifies the number of columns to be occupied by the help window, including the frame, if any. The help will eventually be displayed with one blank column separating the text from each edge of the frame. Thus, the actual width for text is (Width-4) columns. If no !WIDTH directive is specified, MAKEHELP assumes Width=40.

Because MAKEHELP word-wraps the text, the width specification for a help system is built into the help file. The width cannot be changed when the help is later displayed. By contrast, MAKEHELP does not automatically insert page breaks into the help text, thus allowing help screens that can automatically adjust between 25, 43, and 50 line text modes.

The !WIDTH directive must appear before the first !TOPIC directive.

Example
```
!WIDTH 70
```

MAKEHELP's word-wrap will leave a maximum of 66 characters on each line of help text.

!NOWRAP
!WRAP
!NOWRAP disables the word wrapping that MAKEHELP normally performs. Topic help lines are transferred directly from the input text file to the binary help file when wrapping is off. !WRAP turns wrapping on again. Wrapping is on by default. Use the !NOWRAP directive if you've formatted the help text exactly as you want it before running MAKEHELP.

!TOPIC Number [Name]
Each help topic must begin with a !TOPIC directive. Typically, help topics will be entered in ascending numerical order, starting with Number 1. The !TOPIC directive may also specify an optional topic name. The name may contain spaces, and it will later appear on-screen exactly as entered in the !TOPIC directive. (The topic name appears as the title of the help window, and optionally as an entry in the topic index.)

The total space for Names in the help file is limited to 64K bytes. Space used for names is equal to the length of the longest name plus one times the total number of help topics. When topic names are used, this restriction will probably set the limit on the maximum number of topics in a help file. MAKEHELP assures that the 64K limit is not exceeded.

Lines (even blank ones) immediately following the !TOPIC directive start the help text for that topic. The topic continues until another !TOPIC directive is found or the end of the TXT file is reached.

Each help file must have at least one help !TOPIC. Topic number 0 is reserved by the help system to indicate errors, so you should never specify !TOPIC 0.

Examples

```
!TOPIC 1 First Help Section
!TOPIC 22
```

Topic 1 has a name, which will appear as the title of the help window when that topic is displayed. The top border of the help window for topic 22 will be empty, and that topic will not appear in any help index.

!INDEX IndexNumber
!NOINDEX

These directives control whether particular topics will appear in the topic index, and if so, where within the index. The !INDEX and !NOINDEX directives should appear on the line following each !TOPIC directive.

These directives are optional. If they are not used, MAKEHELP indexes each named topic, and sorts the index alphabetically by topic name.

Exclude any named topic from the index by specifying the !NOINDEX directive.

Change the order in which indexed topics appear by specifying the !INDEX directive followed by an integer IndexNumber. Low values of IndexNumber sort to the beginning of the topic index, and high values to the end, overriding the normal alphabetical order. More specifically, the topic index appears in the following order:

```
Topics with IndexNumber < $7777 in ascending numeric order
Topics without !INDEX directives in ascending alphabetic order
Topics with IndexNumber > $7777 in ascending numeric order
```

As you might guess, MAKEHELP assigns a default IndexNumber of $7777 (30583 decimal) to each topic, and the !INDEX directive overrides this value. When MAKEHELP sorts the topics for the help index, it uses IndexNumber as a primary key, and the alphabetic topic name as a secondary key.

Do not specify an IndexNumber of $FFFF (65535 decimal) since MAKEHELP uses this special value to disable indexing for a topic.

Examples

```
!TOPIC 1 Help on Help
!INDEX 1
...
!TOPIC 2 A Very Minor Subtopic
!NOINDEX
...
!TOPIC 3 An Alphabetic Approach
...
!TOPIC 4 The Last Word
!INDEX 32000
...
```

We want the "Help on Help" topic to be listed first in the topic index, so we follow its !TOPIC directive with an !INDEX directive specifying item number 1.

Topic 2, describing a very minor subtopic, should not appear on the index at all, so we remove it with the !NOINDEX directive.

Topic 3 should appear in the help index in normal alphabetic order, so it needs no !INDEX directive.

Topic 4 should be the last topic in the help index, so we give it a large IndexNumber.

!LINE

This directive forces MAKEHELP to finish the current line of text, inserting a line break. Since MAKEHELP automatically fills and word-wraps the text in your original help file, the line breaks in the final help screens are not necessarily the same as those in the original text file. The !LINE command gives you appropriate control when needed. MAKEHELP also inserts a line break whenever it encounters a blank line or a line that begins with one or more spaces.

!PAGE

This directive forces MAKEHELP to finish the current page of text, inserting a page break. Normally, TPHELP paginates the help text only when the help is displayed. For best appearance, however, you may wish to force page breaks at selected locations in your help file.

You may enter comments in the text file--they won't appear in the final help. Comments are indicated when a semicolon appears in the first column of any line.

Example

```
;ANY TEXT APPEARING AFTER THE SEMICOLON IS IGNORED.
```

!INCLUDE IncludeFileName
The specified file is read into the help text stream at this point. The include file capability allows for much larger help systems without using a virtual text editor to create them.

MAKEHELP is configured to allow only one level of include file nesting (include files cannot themselves use the !INCLUDE directive). Nesting can easily be increased by changing a constant (MaxIncludeNest) in the MAKEHELP.PAS source code.

Example

```
!WIDTH 50
!NOWRAP
!INCLUDE HELP.TX1
!INCLUDE HELP.TX2
```

This might be the complete top level TXT file for a particular help system. All of the topics are contained in the two include files HELP.TX1 and HELP.TX2.

Help File Control Characters
Special control characters provide another method to customize the appearance of help displays. With these, you can activate special video attributes; or you can specify that certain passages cross-reference other help topics.

There are three characters that control special video attributes:

```
^A  toggles special attribute 1   Colors[SpAtt1]
^B  toggles special attribute 2   Colors[SpAtt2]
^C  toggles special attribute 3   Colors[SpAtt3]
```

The Colors array specified when you open the help system determines the actual attributes associated with each toggle.

Place the control characters anywhere within the help text (not within help directives, however). The characters themselves will not be displayed and do not affect the length of lines as far as word-wrapping is concerned. The first appearance of a control character causes the following text to appear in the associated color. The next appearance of the same character toggles that color off, returning the active color to its value prior to the previous control character. Use of these toggles may be nested, up to a maximum of 6 levels. The basic logic is the same as that of the FlexWrite routine provided by the TPCRT unit.

To enter control characters using the Turbo Pascal editor, first press <CtrlP>, then hold down the Ctrl key and press the desired character. For example, to enter a ^A into the text, press <CtrlP><CtrlA>.

Two additional control characters activate the cross-reference capability of the help system. A cross-reference is a passage of text that refers to another topic within the help system. While help is displayed, the user may position a highlight bar over the cross-reference passage by pressing the cursor keys. The user can then press <Enter> to select the referenced topic and display its help, and thus browse through many related topics. Any help topic can define up to 50 cross-references.

Control characters within the help text must therefore provide two cues to TPHELP. First, they must indicate which help topic will be activated when a given cross-reference is selected. Second, they must denote the extent of the highlighted passage of text. TPHELP uses the following scheme for this purpose:

```
^Dnnnn^Etext^E
```

The control character ^D precedes the referenced topic number "nnnn", which is expressed using decimal digits. The topic number is terminated by ^E, which is immediately followed by the text passage to be highlighted. A second ^E terminates the highlighted text. Note that the ^E characters must be balanced while the ^D appears alone. Here's an example of two cross-references:

```
^D10^ECompiling^E the help text is easy using ^D5^EMAKEHELP^E.
```

In the real help text, the characters shown with carets will be real control characters, of course. If the user positioned the highlight bar over the word MAKEHELP and pressed <Enter>, the help for topic number 5 would appear.

Text marked with ^E is displayed in either of two attributes. When the cross-reference has been selected, the text appears in the attribute Colors[XsAttr]. When the cross-reference is not selected, the text appears in attribute Colors[XrAttr].

Zero has a special meaning when used as the topic number in a cross-reference. After selecting this cross-reference, instead of another topic, the master topic index is displayed (and the help window switches to pick mode). The help window provides the same behavior as if the user pressed <F1>, but this special cross-reference makes it easier for you to remind people of the master index.

The demonstration program ENTRY.EXE supplied with Turbo Professional uses the TPHELP system. Take a look at ENTRY.TXT for a complete example of a help text file.

Running MAKEHELP

MAKEHELP is a command-line driven utility that reads a text file you've created according to the rules just described, and writes a binary help file ready for use by your program. You call MAKEHELP from the DOS command line as follows:

MAKEHELP HelpTextFile [HelpBinaryFile]

MAKEHELP will read from the text file named by HelpTextFile and will create a new binary file HelpBinaryFile. If HelpBinaryFile is not specified, MAKEHELP will use the text file name after forcing the extension .HLP onto it. MAKEHELP will overwrite any existing HelpBinaryFile without warning. The input file and output file must have different names. For example,

MAKEHELP DEMO.TXT DEMO.HLP
MAKEHELP DEMO.TXT

These two commands do the same thing.

MAKEHELP makes two passes through the input text: one to count various essential items, such as the number of topics; and a second to format the text and write compressed output.

If MAKEHELP finds any errors, it will halt and report the error type and line number within your help text file. MAKEHELP will report a fatal error in the following situations:

Disk I/O problems
Assure that the drives are ready and there is adequate disk space for storage of the output file.

Topic name array exceeds 64K
Use shorter topic names or reduce the number of topics in the help file.

Insufficient memory
MAKEHELP doesn't have enough memory available for temporary storage. Free up some memory by getting rid of TSRs or reduce the size of the help file.

File index array exceeds 64K
The number of topics exceeds 10000. Reduce the number of topics in the help file.

No help topics specified
There were no !TOPIC directives in the input file.

Invalid cross-reference topic number
The number following a ^D character is not valid.

Invalid width
Invalid item
Invalid index
The number following a help directive is not valid.

Duplicate help topic number
The same index was assigned to more than one help topic.

WIDTH statement must precede first help topic
The !WIDTH directive must be the first directive in a help file.

INDEX statement must follow TOPIC statement
Each !INDEX or !NOINDEX directive must follow the !TOPIC directive it refers to.

Unrecognized directive
Column 1 of the help file contains an exclamation point, but the text immediately following it is not a valid help directive.

In one additional situation MAKEHELP reports a warning and continues processing the help file. The warning occurs when a passage of text marked as a cross-reference is split across more than one line of text. As long as these lines are displayed within a single page of help, there is no problem. However, if the lines straddle two pages of help, the portion of the text on the second page will not be selectable by the user. This is not a true problem, but may lead to confusion in some instances. If desired, modify the help text so that the cross-reference is not split across multiple lines.

After successfully processing a help file, MAKEHELP reports some statistics: how many topics there are; how many topics have titles; the size of the largest topic (which determines how large the runtime help buffer must be); and how much MAKEHELP was able to compress the help text.

Binding Help into a Program
TPHELP can directly use the binary file produced by MAKEHELP, without further processing. However, if the help file is relatively small (definitely less than 64K), you may want to link it into your application so that no additional files are required at runtime.

You can do this with the aid of the BINOBJ.EXE utility that Borland supplies with Turbo Pascal. BINOBJ converts the binary help file into a linkable OBJ file that can be bound into your program.

Let's use an example to show how this works. Assume you have a help text file named MYHELP.TXT. The first step would be to run MAKEHELP to create the binary help file:

MAKEHELP MYHELP.TXT MYHELP.HLP

Then run BINOBJ to create the OBJ file:

BINOBJ MYHELP.HLP MYHELP.OBJ MYHELP

The third parameter to BINOBJ specifies the name of the public identifier by which your program can refer to the contents of the OBJ file.

Now, within your application program, insert the following lines of source code:

```
{$L MYHELP.OBJ}
procedure MyHelp; external;
```

The $L compiler directive tells Turbo to link in MYHELP.OBJ. The fake procedure MyHelp gives your program a way to refer to the contents of that OBJ file.

Finally, to initialize the help system, make a call to OpenHelpMem as follows:

```
Status := OpenHelpMem(@MyHelp,
                      LeftCol, TopRow, BotRow, PickCols,
                      Colors, Help);
if Status <> 0 then
  {Error, help not available};
```

The notation "@MyHelp" passes the address of the binary structure to the help system initialization procedure.

From then on, simply refer to the Help variable returned by OpenHelpMem just as you would refer to one initialized by OpenHelpFile.

Types

```
HelpColorType = (FrAttr,  {Color for window frame}
                 TeAttr,  {Color for normal help text}
                 HeAttr,  {Color for window header}
                 XsAttr,  {Color for selected cross-reference item}
                 XrAttr,  {Color for unselected cross-reference item}
                 SpAtt1,  {Color for ^A help text}
                 SpAtt2,  {Color for ^B help text}
                 SpAtt3); {Color for ^C help text}
HelpColorArray = array[HelpColorType] of Byte;
```

HelpColorType defines the various colors used within a help window. The HelpColorArray is just a convenient way to collect these colors into a single structure. See the preceding discussion for more details about activating the various attributes.

```
HelpPtr = ^HelpDesc;
```

A variable of this type is the main hook between your program and the help system. TPHELP initializes a variable of this type when you open a help file, by allocating heap space where information describing the help system is stored. See TPHELP.PAS for further details about the HelpDesc type.

```
HKtype = HKSNone..HKSUser3;
```

HKType defines the class of actions performed by the help system. It includes such actions as cursor up and down, page up and down, display help index, and escape. See the constants below for further details.

Constants

```
MaxPagesPerSection = 20;
MaxXrefsPerSection = 50;
MaxTopics = 10000;
MaxHelpStack = 19;
```

These constants specify the capacity of the help system, and are largely self-explanatory. MaxHelpStack determines the number of previous help topics that TPHELP can remember--these previously displayed topics may be recalled by pressing <AltF1> while within the help system. With the exception of MaxTopics, changes in these constants directly affect the heap and stack usage of TPHELP. (The space used for topics depends on the help file itself, and isn't directly affected by the value of MaxTopics.) You may increase or reduce these values and recompile TPHELP to trade off capacity with memory usage.

```
Attr1Toggle = ^A;
Attr2Toggle = ^B;
Attr3Toggle = ^C;
IndexMarker = ^D;
XrefToggle = ^E;
LineBrkMark = ^M;
PageBrkMark = ^L;
SectEndMark = #0;
```

These constants specify characters used within the help text to denote special formatting. The control characters ^A..^E have been discussed in a previous

section. ^M marks the end of each wrapped line of help text, and ^L indicates a forced page break. Each help topic ends with a section marker #0. In the binary help file, these markers are stored in compressed format, so you probably won't recognize them.

```
HelpId : array[0..3] of Char = 'TPH2';
```

Every binary help file starts with a sequence of characters to denote the proper format. This constant specifies the current format. Note that the help file format in this version of Turbo Professional is significantly different than that in earlier versions of TPHELP. To use earlier help files, you'll need to recompile them with the current version of MAKEHELP.

```
HKSNone = 0;        {Not a command}
HKSAlpha = 1;       {An alphanumeric character, ignored}
HKSUp = 2;          {Cursor up to previous cross-reference}
HKSDown = 3;        {Cursor down to next cross-reference}
HKSPgUp = 4;        {Display previous help page}
HKSPgDn = 5;        {Display next help page}
HKSLeft = 6;        {Cursor left to previous cross-reference}
HKSRight = 7;       {Cursor right to next cross-reference}
HKSExit = 8;        {Exit the help system}
HKSSelect = 9;      {Select the current cross-reference and display topic}
HKSPrev = 10;       {Display the most recent help topic}
HKSHome = 11;       {Display first help page}
HKSEnd = 12;        {Display last help page}
HKSIndex = 13;      {Display an index of all help topics}
HKSProbe = 14;      {Check the mouse position and perform an action}
HKSUser0 = 15;      {User-defined exit commands}
...
HKSUser3 = 18;
```

These are the codes for each of TPHELP's cursor movement and selection commands. Most are self-explanatory. The last four (HKSUser0..HKSUser3) are set aside to allow user-defined "exit commands" to be implemented. If one of these commands is executed, TPHELP will do the same things that it would if <Esc> were pressed; it will then be up to you to check the global variable HelpCmdNum and perform the function associated with the command just issued. For more details on how to implement user-defined commands, see Appendix B.

```
HelpTitle : string[39] = ' Topics ';
```

TPHELP displays this string at the top of the help window whenever it has drawn a pick list of help topics.

```
UseHelpFrame : Boolean = True;
```

Set this constant False prior to calling OpenHelpFile or OpenHelpMem if you don't want a frame drawn around the help window.

```
HelpMore : Boolean = True;
```

Set this constant False prior to calling OpenHelpFile or OpenHelpMem if you don't want PgUp/PgDn prompts to appear in the help window frame when more pages of help are available.

```
HideCursor : Boolean = True;
```

Set this constant False prior to calling any help display function if you want the blinking hardware cursor to remain visible while the help window is open.

```
HelpMouseScroll : Boolean = True;
```

When TPHELP's mouse capabilities are enabled (by calling EnableHelpMouse), this constant controls whether TPHELP displays and supports a scroll bar. Since the scroll bar appears within the right hand frame of the help window, it is only available when the window is framed. Beyond that, the scroll bar appears only when the current help topic consists of more than one page.

The scroll bar has several capabilities. At the upper and lower corners of the frame, vertical arrows are displayed. TPHELP scrolls the item display by one page whenever the mouse cursor has been positioned over one of these arrows and the left button pressed. TPHELP also displays a block character within the vertical frame to indicate the approximate position of the current page within the topic. TPHELP displays a new page and adjusts this "slider" when the mouse cursor is placed at a different location within the scroll bar and the left button pressed.

Note that you must set the value of HelpMouseScroll prior to calling OpenHelpFile or OpenHelpMem if you want the change to take effect.

Variables

```
HelpKeyPtr : Pointer;
```

Points to a routine that will return the next keystroke. See Appendix B for details.

```
HelpCmdNum : HKtype;
```

The last help command given to TPHELP. Useful only if you have user-defined exit commands that require special processing after the help window is closed.

CloseHelp / DisableHelpIndex

Declaration

```
procedure CloseHelp(var Help : HelpPtr);
```

Purpose

Close help file and/or deallocate buffer space.

Comments

CloseHelp closes the help file, if one was opened, and deallocates all buffer space for a help system. The Help parameter must have been initialized previously by OpenHelpFile or OpenHelpMem. After a call to CloseHelp, Help will be set to nil.

Example

```
var Help : HelpPtr;
...
Status := OpenHelpFile('DEMO.HLP', 10, 2, 22, 2, HelpColors, Help);
CloseHelp(Help);
```

Opens and immediately closes the help file DEMO.HLP.

See Also

OpenHelpFile

OpenHelpMem

Declaration

```
procedure DisableHelpIndex;
```

Purpose

Disable the <F1> help index inside of the help window.

Comments

Normally, when a help window is popped up and the user presses <F1>, TPHELP automatically displays the master index of all help topics. From this point, the user can select any other indexed topic and select help for that item. Calling DisableHelpIndex at any point in a program disables the use of <F1> within the popup help window. The help index can be reenabled by calling EnableHelpIndex.

DisableHelpIndex is a special case of dynamically modifying the keyboard interface of the TPHELP unit. See Appendix B for further details.

Example

```
Status := OpenHelpFile('DEMO.HLP', 10, 2, 22, 2, HelpColors, Help);
DisableHelpIndex;
if ShowHelp(Help, 1) then ;
EnableHelpIndex;
if ShowHelp(Help, 1) then ;
```

The first call to ShowHelp will not allow the use of the <F1> key within the popup window. The second call will.

See Also

EnableHelpIndex

DisableHelpMouse / EnableHelpIndex

Declaration

```
procedure DisableHelpMouse;
```

Purpose

Disable mouse control of help screens.

Comments

This routine undoes the actions of EnableHelpMouse. For many programs, you'll never need to call DisableHelpMouse, since its essential functions are handled automatically by a Turbo Pascal exit routine. In some cases, particularly in memory resident programs, you may need to call DisableHelpMouse in order to disable the mouse functions temporarily.

Note that DisableHelpMouse does not hide the mouse cursor. You can call HideMouse to do that if appropriate.

Examples

```
{TSR pops up}
...
EnableHelpMouse;
...
if ShowHelp(...) then begin
  ...
end;
DisableHelpMouse;
...
{TSR returns}
```

See Also

EnableHelpMouse

Declaration

```
procedure EnableHelpIndex;
```

Purpose

Enable the <F1> help index inside of the help window.

Comments

This procedure creates the opposite effect of the DisableHelpIndex procedure by reenabling the use of the <F1> key within a popup help window. See DisableHelpIndex for details.

See Also

DisableHelpIndex

EnableHelpMouse

Declaration

```
procedure EnableHelpMouse;
```

Purpose

Enable mouse control of help screens.

Comments

Call this routine to activate mouse control of the help system. The routine checks to assure that a mouse is installed, then sets an internal flag so that other pick routines will utilize the mouse. You should call EnableHelpMouse once prior to calling TPHELP's HelpWindow routine.

Since TPHELP's mouse control depends on the TPMOUSE keyboard event handler, EnableHelpMouse also calls TPMOUSE's EnableEventHandling routine. There is no harm done if you've called this routine previously. Be aware, though, that any other mouse event handler which you've previously activated will lose control after the call to EnableHelpMouse.

EnableHelpMouse does not modify the current mouse window, the mouse cursor type, or the mickey-to-pixel ratio, nor does it display the mouse cursor. Your program should set those values as needed by calling the appropriate TPMOUSE routines. You may use either a hardware or software mouse cursor.

When the mouse is enabled, TPHELP performs a number of added functions while help screens are displayed. See procedure ShowHelp for details.

If the help window has a frame and the TPHELP typed constant HelpMouseScroll is True, the mouse can also be used to select among the pages of each topic.

The actions performed when mouse buttons are activated may be configured in the same way that keystrokes are installed. See Appendix B for details.

Note that mouse actions occur when the buttons are *released*, not when they are first pressed. This technique allows reliable use of commands that require both buttons to be pressed simultaneously. Change this behavior by toggling the WaitForButtonRelease constant in TPMOUSE. After that change, mouse actions occur immediately after a mouse button is depressed.

In a memory resident program using TPHELP, you should not enable the mouse prior to going resident. Instead, you should enable the mouse after popping up, and disable it again before returning. There are additional considerations when writing a memory resident program that uses the mouse. See Chapter 2, Section G for details.

Examples

```
begin
  ClrScr;

  TpHelp.HelpMouseScroll := True;
  EnableHelpMouse;

  Status := OpenHelpFile('DEMO.HLP',    {Name of binary help file}
                         10, 2,         {Upper left corner of help window}
                         22, 2,         {Lower row, and columns in help index}
                         HelpColors,    {Colors to use}
                         Help);         {Help descriptor}
  if Status <> 0 then
    {Error} ;

  TpWindow.Shadow := True;
  TpWindow.ShadowAttr := $70;
  repeat
    Topic := PickHelp(Help, 2, 2, 10, 1);
    if Topic <> 0 then
      if not ShowHelp(Help, Topic) then
        WriteLn('Unable to display help for topic ', Topic);
  until Topic = 0;

  CloseHelp(Help);
end.
```

Shows the modifications required to the introductory example to provide mouse support within the help system.

See Also

DisableHelpMouse

ShowHelp

FindHelp

Declaration

```
function FindHelp(Help : HelpPtr; Name : string; MatchFunc : Pointer) : Word;
```

Purpose

Return topic number of help with specified Name, 0 if not found.

Comments

FindHelp searches all of the topic names for the specified help system to find one that matches the parameter Name. By default, searching is not case sensitive and a topic name will match only if there is a complete match between the specified name and the actual topic. The Help parameter to FindHelp must have been initialized previously by calling OpenHelpFile or OpenHelpMem.

The default search behavior is obtained when the third parameter, MatchFunc, is nil. If the address of another function is passed in this parameter, FindHelp will use an alternate match function. The function whose address is passed must be compiled under the FAR model, and it must match the following definition:

```
{$F+}
function Matches(TopicName, Name : string) : Boolean;
{$F-}
```

This function should return True if the two strings are considered equal. The first parameter is the topic currently being tested, and the second parameter is the parameter passed to FindHelp, which will be the same for each call.

When a match is found, FindHelp returns the index of the topic that matched. Otherwise, FindHelp returns zero. Any non-zero index returned by FindHelp may be used as a parameter to ShowHelp in order to display help about that topic.

Example

```
{$F+}
function Matches(TName, Name : string) : Boolean;
begin
  {Match if the topic name begins with the sought name}
  Matches := Pos(StUpcase(Name), StUpcase(Trim(TName))) = 1;
end;
{$F-}

begin
  Topic := FindHelp(Help, 'COMPIL', @Matches);
end.
```

FindHelp returns the index of any topic whose name begins with the letters 'COMPIL', thus matching topics like ' Compilation', 'Compiler', and so on. Note that the user-defined match function is forced to compile FAR, and that it raises each string to uppercase and trims blanks from the help topic name before performing the comparison.

See Also

ScreenHelp

Declaration

```
function OpenHelpFile(HelpFileName : string;
                      XLow, YLow, YHigh, PickCols : Byte;
                      Colors : HelpColorArray;
                      var Help : HelpPtr) : Word;
```

Purpose

Find and open help file, returning initialized help descriptor and a status code.

Comments

OpenHelpFile first attempts to find the help file specified in the first parameter. It uses the ExistOnPath function from the TPDOS unit to do so. If HelpFileName contains a path specification, only the directory of that path is checked. If HelpFileName is a bare file name, ExistOnPath looks in first the current directory, then the directory where the program itself was loaded (if running under DOS 3.0 or later), and finally in all directories on the DOS path. If the file is not found, OpenHelpFile exits with an error.

Next, OpenHelpFile reads the file header to validate that the file is in the proper format. If not, it exits with an error.

Then OpenHelpFile allocates a buffer large enough to hold the largest single help topic once it has been decompressed. If insufficient heap space, it exits with an error.

Finally, OpenHelpFile validates the number of pick columns, and reduces the requested value if the topic names are too large to fit within the help window. Remember, the pick columns come into use only when the <F1> key is pressed within a popup help window to display a master help index.

OpenHelpFile stores various parameters within the record pointed to by the returned Help variable. Besides the help window coordinates and colors, it also stores the current value of UseHelpFrame, HelpMore, and HelpMouseScroll, global typed constants described previously.

OpenHelpFile leaves the physical help file open, consuming one DOS file handle. Except for a small header, no part of the help file is read until required by a request to display a help topic. Do not call OpenHelpFile more than once for each help system unless the help system has been closed by a call to CloseHelp.

OpenHelpFile returns Turbo Pascal runtime error codes in its function result. Here is a list of the most common error codes:

```
  0   Success
  2   File not found, or out of file handles
100   Unexpected end of file reading help
106   Invalid file format
203   Insufficient heap space for help buffers
```

For small help files, you may find it more convenient to permanently bind the help information into your program. See OpenHelpMem for details.

OpenHelpFile

Example

```
Status := OpenHelpFile('C:\MYAPP\DEMO.HLP', 10, 2, 22, 2, HelpColors, Help);
```

OpenHelpFile looks only in directory C:\MYAPP to find the help file.

See the example at the beginning of this section for a complete working program that calls OpenHelpFile.

See Also

OpenHelpMem

Declaration

```
function OpenHelpMem(HPtr : Pointer;
                     XLow, YLow, YHigh, PickCols : Byte;
                     Colors : HelpColorArray;
                     var Help : HelpPtr) : Word;
```

Purpose

Initialize the help descriptor for a help file bound into a program, returning a status code.

Comments

OpenHelpMem is an alternative method of initializing a help system. In contrast to OpenHelpFile, which leaves a help file open and causes the help system to read from the disk every time a help topic is displayed, OpenHelpMem just initializes some pointers to information which is already in memory. As a result, help system response times will be shorter with OpenHelpMem, and the user won't need to worry about copying around a separate help file. On the other hand, the size of the application's EXE file will increase when the OpenHelpMem approach is used.

To prepare for using OpenHelpMem, you must bind the help file into your program. See the section "Binding Help into a Program" earlier in this chapter for more details. Once this is done, you simply refer to the address of the bound-in help "procedure" by using Turbo Pascal's '@' operator:

```
Status := OpenHelpMem(@MyHelp, XL, YL, YH, PickCols, Colors, Help);
```

Since the dummy procedure associated with your bound-in help file is never actually called, it does not matter whether it is compiled under the NEAR or FAR model.

OpenHelpMem returns status codes just like OpenHelpFile, with the exception that the "file not found" error will never occur. Like OpenHelpFile, OpenHelpMem must allocate a buffer on the heap to hold the largest topic's uncompressed text.

Example

```
program HelpDemo;
uses
  Dos, TpDos, TpString, TpCrt, TpWindow, TpCmd, TpPick, TpHelp;
const
  HelpColors : HelpColorArray = ($07, $07, $0F, $70, $09, $70, $0F, $01);
var
  Help : HelpPtr;
  Status : Word;
  Topic : Word;

{$L DEMOHELP.OBJ}
procedure DemoHelp; external;
```

```
begin
  ClrScr;
  Status := OpenHelpMem(@DemoHelp, 10, 2, 22, 2, HelpColors, Help);
  if Status <> 0 then begin
    WriteLn('Help initialization error ', Status);
    Halt;
  end;

  repeat
    {Pick a topic from the help index}
    Topic := PickHelp(Help, 2, 2, 10, 1);
    if Topic <> 0 then
      {Display help for that topic}
      if not ShowHelp(Help, Topic) then
        WriteLn('Unable to display help for topic ', Topic);
  until Topic = 0;

  {Release help memory}
  CloseHelp(Help);
end.
```

Performs the same functions as the example at the beginning of this section, using bound-in help rather than a file.

See Also

OpenHelpFile

Declaration

```
function PickHelp(Help : HelpPtr; XLow, YLow, YHigh, PickCols : Byte) : Word;
```

Purpose

Display the help index, returning the selected topic number, 0 if none.

Comments

Call this routine to display a pick list of available help topics. This works just like the help index that appears when a user presses <F1> while within a help window, with two exceptions. First, the coordinates of PickHelp's window are set independently by the parameters you pass to it, unlike the internal pick list, whose coordinates match that of the help window itself. Second, PickHelp returns a topic index number instead of immediately displaying the help.

The Help parameter must have been initialized previously by a call to OpenHelpFile or OpenHelpMem. XLow and YLow specify the upper left corner of the pick window, including frame if any. (The setting of the global constant UseHelpFrame determines whether the pick window has a frame.) YHigh is the bottom row of the pick window, adjusted according to the conventions of the PickWindow routine. And PickCols is the number of columns of help items to display side-by-side.

PickHelp must allocate two temporary buffers on the heap. One buffer holds all of the help topic names, and may be as large as 64K depending on the size of the help system. The other holds a sorted topic map, which indicates the order in which the help index will appear, and uses two bytes for each help topic. If insufficient heap space is available, PickHelp will immediately exit, returning a topic number of 0.

PickHelp then displays the pick list and lets the user choose a topic. If the user presses <Esc>, PickHelp returns topic number 0. Otherwise, it returns the selected topic number.

If mouse support is enabled for TPHELP, PickHelp temporarily enables mouse support for TPPICK as well.

Example

```
TpHelp.UseHelpFrame := False;
TpPick.PickSrch := StringPickSrch;
TpWindow.Explode := True;
Topic := PickHelp(Help, 2, 2, 10, 1);
if Topic <> 0 then
  if not ShowHelp(Help, Topic) then
    WriteLn('Unable to display help for topic ', Topic);
```

This shows how you can use the various modes of units on which TPHELP depends to adjust the appearance of the pick window. This particular window will have no frame, will support incremental string searching, and will explode when first displayed. Its upper left corner will be at (2,2), it will show a maximum of 9 rows of topics, one across.

See Also

PickWindow (TPPICK) ShowHelp

ScreenHelp

Declaration

```
function ScreenHelp(Help : HelpPtr; XPos, YPos : Byte;
                    ScanBack : Boolean; MatchFunc : Pointer) : Word;
```

Purpose

Return topic number whose name matches screen contents at position (XPos,YPos), 0 if none.

Comments

You use this function to offer help based on screen lookup. For example, suppose a programmer wants information about a particular procedure, and the editor's cursor is currently within the procedure name. ScreenHelp could parse the word nearest the cursor, search the help system for a topic matching that name, and return the topic number, if any.

The Help parameter passed to ScreenHelp must have been initialized by a previous call to OpenHelpFile or OpenHelpMem. XPos and YPos are the absolute screen coordinates where ScreenHelp should look for a word. If ScanBack is True, ScreenHelp will scan from the starting XPos to the left to skip over any characters not considered part of a word. Consider the following example:

```
It rained on our trip to Spain.      _
```

If ScanBack were False, ScreenHelp would notice that the cursor is not within a word and immediately exit without further searching. If ScanBack were True, ScreenHelp would scan the screen backward, starting just to the left of the cursor, skip over the spaces and the period, parse the word Spain from the end of the line, and then search for help regarding "Spain."

When parsing words from the screen, ScreenHelp considers the following characters to be valid parts of a word:

```
'A'..'Z', '_', '0'..'9'
```

MatchFunc specifies the address of a function compiled under the FAR model that determines whether the word parsed from the screen matches a given help topic name. If you pass a value of nil for this parameter, ScreenHelp uses its default match function, which requires that the parsed word and the topic name match exactly (except for case, which doesn't matter). You can write a routine that allows for approximate matches, perhaps requiring just that the word from the screen is contained anywhere within the help topic name. Your match function must be declared like the following prototype:

```
{$F+}
function Matches(TopicName, ScreenName : string) : Boolean;
{$F-}
```

This function should return True if the two strings are considered equal. The first parameter is the topic currently being tested, and the second parameter is the word parsed from the screen, which will be the same for each call.

When a match is found, ScreenHelp returns the index of the topic which matched. Otherwise, ScreenHelp returns a value of zero. Any non-zero index returned by

ScreenHelp may be used as a parameter to ShowHelp in order to display help about that topic.

Example

```
{$F+}
function Matches(TName, SName : string) : Boolean;
begin
  {Match if the screen word is contained anywhere within the topic name}
  Matches := Pos(StUpcase(SName), StUpcase(TName)) <> 0;
end;
{$F-}


begin
  Topic := ScreenHelp(Help, WhereXAbs, WhereYAbs, True, @Matches);
end.
```

ScreenHelp takes the word nearest the hardware cursor's position on-screen, and returns any topic whose name contains that word.

See Also

FindHelp

SetHelpPos

Declaration

```
procedure SetHelpPos(Help : HelpPtr; XLow, YLow, YHigh : Byte);
```

Purpose

Change the position and height of a help window after the Help descriptor has already been initialized.

Comments

When you first open a help file, you specify the position at which the help window will appear. This procedure allows you to change that position later on. Note that the routine cannot be called while the help window is currently displayed.

Example

```
var
  Help : HelpPtr;
begin
  Status := OpenHelpFile('DEMO.HLP', 10, 2, 22, 2, HelpColors, Help);
  ...
  SetHelpPos(Help, 15, 5, 22);
end;
```

The call to SetHelpPos moves the help window down and to the right, making the window smaller at the same time.

Declaration

```
function ShowHelp(Help : HelpPtr; Topic : Word) : Boolean;
```

Purpose

Display help screen for topic, returning True if successful.

Comments

This is the primary routine for displaying help. Given an initialized Help variable and a valid Topic number, ShowHelp pops the help window onto the screen and displays the first help page for the topic. From there, the user can perform a number of actions, including the following:

<PgUp>, <PgDn>
Display the previous or next help page.

<Esc>
Exit the help window, restoring the screen to its previous state.

<Up>, <Down>, <Left>, <Right>
Move the highlight bar among the cross-reference topics, if any, displayed on the current help page.

<Enter>
Select the currently highlighted cross-reference topic, and display the first help page for that topic.

<F1>
Display an index of all help topics, and let the user choose a new topic by using the cursor keys and pressing <Enter>. If the user presses <Esc> while in the index, the help window is erased and ShowHelp exits. If the user presses <AltF1> while in the index, the previous help topic is restored.

<AltF1>
Display the help topic most recently selected. ShowHelp maintains a stack of the 19 most recently selected help topics. Each time another topic is selected, ShowHelp pushes the previous topic onto this stack. Each time the user presses <AltF1>, the top element of this stack is retrieved and the corresponding help displayed.

When mouse support has been enabled by calling EnableHelpMouse, and the mouse scroll bar has been activated by setting HelpMouseScroll True (the default), then ShowHelp takes advantage of the mouse in several ways. First, the right mouse button is equivalent to pressing <Esc>--it exits the help system. Second, pressing both mouse buttons is equivalent to keying <F1> within the help system--it brings up the help index.

The mouse may also be used to select cross-reference topics. Positioning the mouse cursor over a cross-reference and pressing the left button moves the highlight. Pressing the left button a second time selects the topic and displays a new help screen, just like pressing the <Enter> key.

In the upper left corner of the help window is a small diamond character. Positioning the mouse cursor there and pressing the left button redisplays the most recently selected topic, just like pressing <AltF1>.

The right edge of the help window contains a scroll bar. (If the help window isn't framed, or if there is only one page of help in the topic, the scroll bar won't appear.) Positioning the mouse cursor over the arrow characters at the top or bottom of the scroll bar and pressing the left button causes ShowHelp to display the previous or next page of help, respectively. Within the central portion of the scroll bar, TPHELP displays a block character called the "slider." The position of the slider indicates the relative position of the current help page within the topic. Positioning the mouse cursor above or below the slider and pressing the left button causes TPHELP to display a new help page, its position proportional to the mouse cursor's position.

Any of the commands may be disabled or assigned new keystrokes. See Appendix B for details.

If an error occurs while attempting to display help, ShowHelp returns False. Errors may result from insufficient memory (for a window buffer), a disk error (perhaps because the help file is no longer available), or an invalid topic number (0 or one for which no help exists). In addition, when the user requests a help index, ShowHelp must allocate additional buffers for the topic names. If insufficient heap space is available, ShowHelp simply does not allow the help index, but otherwise continues processing.

ShowHelp depends on TPWINDOW for its window processing. As a result, you can take advantage of TPWINDOW's special modes, such as shadowing and explosions, to adjust the appearance of your help screens.

You can centralize calls to ShowHelp by taking advantage of Turbo Professional's redefinable keyboard handlers. To do so, you would make the following declarations:

```
const
  HelpKey : Word = $3B00; {F1 key}
var
  CurrentTopic : Word;
  Help : HelpPtr;

{$F+}
function ReadKeyHelp : Word;
var
  KW : Word;
begin
  repeat
    {Read the next key from the keyboard}
    KW := TpCrt.ReadKeyWord;
    if KW = HelpKey then
      {Request for help}
      if not ShowHelp(Help, CurrentTopic) then
        Beep;
  until KW <> HelpKey;
  ReadKeyHelp := KW;
end;
{$F-}
```

You would then assign the keyboard handlers of any other units (such as TPEDIT, TPPICK, TPENTRY) to point to this routine. See Appendix B for details. Other portions of your program would keep the variable CurrentTopic up to date with whatever help topic was appropriate. Then, anywhere keyboard input was required, the user could press the help key, and help for the current topic would pop up.

Example

```
if not ShowHelp(Help, 1) then
  Beep;
```

The topic number may be a constant, a global variable holding the current context sensitive topic number, or may be computed based on a topic name or screen lookup.

See Also

DisableHelpIndex

EnableHelpMouse

ShowPrevHelp

Declaration

```
function ShowPrevHelp(Help : HelpPtr) : Boolean;
```

Purpose

Display help screen for topic most recently selected, returning True if successful.

Comments

This routine gives your program direct access to TPHELP's stack of previously displayed help topics. When you call ShowPrevHelp, it displays the previous topic and pops that topic from the stack. If the stack is empty, or if some other error occurs, ShowPrevHelp returns False.

All of the capabilities of ShowHelp are available while ShowPrevHelp is active.

Examples

```
if not ShowPrevHelp(Help) then
  Beep;
```

See Also

ShowHelp

G. TPMOUSE: Mouse Routines

In writing TPMOUSE, we aspired to three main goals. First, we wanted it to provide routines that were optimized for mouse programming in text mode, in part because other collections of mouse routines that we have seen tend to be weak in this area, but also because all the other screen-oriented routines in Turbo Professional are designed to be used in text mode. Second, we wanted these routines to provide a high-level interface, insulating the programmer as much as possible from the mouse driver. And third, we wanted it to provide a body of routines that we could build on in other Turbo Professional units, insulating the programmer still further.

Getting Started with TPMOUSE

As you begin working with TPMOUSE, you should be pleased to learn that we have chosen to model it after TPCRT, a unit that you're probably already familiar with. Besides reducing the learning curve for TPMOUSE, this has some other benefits that we'll be pointing out later. To give you an idea of the extent to which the model has been followed, here's a quick list of rough correspondences:

TPMOUSE routine	TPCRT "equivalent"
MousePressed	KeyPressed
MouseKeyWord	ReadKeyWord
InitializeMouse	ReinitCrt
MouseWhereX	WhereX
MouseWhereY	WhereY
MouseWhereXY	WhereXY
MouseGotoXY	GotoXY
MouseWindow	Window
NormalMouseCursor	NormalCursor
FatMouseCursor	FatCursor
BlockMouseCursor	BlockCursor
HiddenMouseCursor	HiddenCursor
GetMousePage	CurrentPage (global variable)
SetMousePage	SetVisiblePage

With a few significant exceptions, most of the other routines in TPMOUSE are ones that you won't need very often. If you're already up to speed with TPCRT, you should therefore have little difficulty in getting familiar with the basic tools provided in TPMOUSE.

Probably the single most important thing to keep in mind when using the routines in this unit is that most of them should not be called unless the global variable MouseInstalled is True. This rule is not enforced, and you'll get unpredictable results if you break it. The only real exceptions to the rule are InitializeMouse (the routine used to set MouseInstalled in the first place), MousePressed (always returns False if a mouse is not installed), ReadKeyOrButton (always returns a key if MouseInstalled is False), MouseStateBufferSize (always returns 0 if there's no

mouse), HideMouse (does nothing if there is no mouse), and ShowMouse (also does nothing if there is no mouse). Although a few of the other routines will do no damage if MouseInstalled is False, a well-designed program will assume that it is not safe to call any routine other than those listed above unless MouseInstalled is True.

Mouse Support in Other Units

As mentioned earlier, five of the other units in Turbo Professional provide mouse support based on the routines in the TPMOUSE unit: TPEDIT, TPMENU, TPPICK, TPHELP, and TPENTRY. (TPDIR does as well, provided that mouse support is enabled in TPPICK.) In each case, you can enable this support by calling a single routine:

```
EnableEditMouse  (TPEDIT)
EnableMenuMouse  (TPMENU)
EnablePickMouse  (TPPICK)
EnableHelpMouse  (TPHELP)
EnableEntryMouse (TPENTRY)
```

In most cases, all you need to do before calling these routines is to select the kind of mouse cursor that you want to use (see the discussion of "The Mouse Cursor," below). You may also want to make the mouse cursor visible at this point, by calling ShowMouse, but that isn't always necessary. Once mouse support is enabled in these units, the routines in them will make sure that the mouse cursor is on when they need it to be, and restore it to its previous state on exit.

So, although you will usually want to provide additional forms of mouse support in your own code, it is actually possible to write a program that makes effective use of the mouse simply by adding a few procedure calls to the beginning of it. If you examine the demo programs PTRPT.PAS and VBROWSE.PAS, paying special attention to sections of code starting with '{$IFDEF UseMouse}', you'll see that remarkably little effort was required in order to add mouse support to the programs.

The Mouse Cursor

In the previous paragraph, we indicated that you need to choose the kind of mouse cursor that you want to use. If you weren't aware of it before, mouse cursors come in two varieties, hard and soft. When you choose a hardware mouse cursor, the mouse driver simply moves the "real" cursor around the screen in response to the movements of the mouse. (Hereafter, we'll refer to the "real" cursor as "the system cursor".) When you choose a software cursor, the mouse driver leaves the system cursor alone and instead manipulates characters on the screen (temporarily changes their color and/or overwrites them with another character) in order to indicate the position of the mouse cursor. When the mouse is moved, the character beneath the mouse cursor is restored, and another location on the screen is altered.

If you're having a hard time picturing all this, we suggest that you try running two of the programs on the DEMO disk before going any further: MAKEMENU and PREF. MAKEMENU uses a software mouse cursor (shaped like a diamond), because it needs the system cursor for other things. PREF doesn't, so it uses a hardware mouse cursor (a blinking, block-shaped cursor) because doing so can simplify matters a bit, as we'll explain later.

A hardware mouse cursor is the best choice for almost every program in which it is an alternative. Unfortunately, those programs are rare. Most programs that have a use for the mouse also have a use for the system cursor, and can't afford to have it moved each time the user moves the mouse. If you do have a program that fits the bill, however, then the only real decision you have to make is the shape of the cursor. TPMOUSE gives you several options. You can use the HardMouseCursor routine, which lets you specify the starting and ending scan lines for the cursor, or you can use one of the higher level routines: NormalMouseCursor, FatMouseCursor, or BlockMouseCursor.

A word of warning about the hardware mouse cursor. The mouse driver positions it by programming the CRT controller, and it does not update the BIOS data area, so you cannot determine its position using WhereX, WhereY, etc. The only routine in TPCRT that will return the position of a hardware mouse cursor is WhereXYdirect. For related reasons, the cursor positioning routines in TPCRT (GotoXY, GotoXYabs) should not be used to move a hardware mouse cursor.

A software cursor will be your choice not only in cases where use of a hardware cursor isn't feasible, but also in cases where you want to achieve a special effect that isn't possible with a hardware cursor. The default mouse cursor is a sort of non-blinking/see-through/reverse-video block, whose color changes as it moves in order to assure that its color always contrasts with that of the character beneath it. How is this possible?

When you select a software cursor (using SoftMouseCursor), you must specify a "screen mask" and a "cursor mask". The easiest way to understand how these values are used is to study a couple of examples. Let's say that the mouse cursor is about to be moved on top of a character, 'A' (ASCII #$41), which is being displayed in white on black (a video attribute of $0F). In video memory, this character is represented as a word, with the attribute in the high byte: $0F41. Given the default screen mask of $FFFF (= DefaultScreenMask) and the default cursor mask of $7700 (= DefaultCursorMask), the mouse driver would then create the mouse cursor by first ANDing the character value with the screen mask

```
$0F41 AND $FFFF  =  $0F41 <-- no change
```

and then XORing the result with the cursor mask:

```
$0F41 XOR $7700  =  $7841 <-- attribute changed, character unchanged
```

This would produce a letter 'A' displayed in reverse video (dark gray on light gray). Now let's say you wanted a slightly different effect, variable colors with a mouse cursor shaped like a diamond (ASCII #$04). In this case you'd use a screen mask of $FF00 and a cursor mask of $7704:

```
$0F41 AND $FF00  =  $0F00 <-- attribute unchanged, character wiped out
$0F00 XOR $7704  =  $7804 <-- attribute changed, new character in low byte
```

Finally, let's say that you want the mouse cursor to look the same no matter where it is on the screen: you always want want it to appear as a musical note (ASCII #$14) displayed in white on red ($4F). In this case you'd use a screen mask of $0000 and a cursor mask of $4F14:

```
$0F41 AND $0000  =  $0000 <-- attribute and character wiped out
$0000 XOR $4F14  =  $4F14 <-- new attribute, new character
```

Reading The Mouse Buttons

When you start adding mouse support to your programs, you'll soon discover that it really isn't that difficult if you have the right data at your disposal. In most cases, that means that you need to know one or both of two things: where the mouse is, and what mouse button(s) if any have been pressed. In this section, we'll concentrate on the mouse buttons.

As indicated earlier, TPMOUSE is modelled, to some extent, on the TPCRT unit. So it should come as no surprise that two of the most basic functions in TPMOUSE are called MousePressed and MouseKeyWord, which are to the mouse buttons what KeyPressed and ReadKeyWord are to the keyboard. MousePressed tells you if a mouse button (or combination of buttons) is being pressed, and MouseKeyWord returns a value indicating what was pressed. Before using them, there are only two things you need to know. First, these functions can do nothing until EnableEventHandling has been called. We'll explain why in the discussion of "Mouse Event Handlers." Second, MouseKeyWord returns what we call a "pseudo-scan code," a word value that looks just like a scan code returned by ReadKeyWord:

```
MouseLft    = $EF00; {left button}
MouseRt     = $EE00; {right button}
MouseBoth   = $ED00; {both buttons}
MouseCtr    = $EC00; {center button}
MouseLftCtr = $EB00; {left and center buttons}
MouseRtCtr  = $EA00; {right and center buttons}
MouseThree  = $E900; {all three buttons}
```

The low byte is always 0, and the high byte indicates which button or buttons were

pressed. Among other things, this system makes it easy to add mouse support to programs that are already set up to handle keyboard input. (These pseudo-scan codes are displayed by PREF, our memory resident quick reference chart.)

That raises the question, "Wouldn't it make things easier if there were a single function that returned either a key or a mouse button?" Yes indeed, and the function that does it is ReadKeyOrButton. This routine simply sits in a loop waiting for a key or mouse button to be pressed, then returns either the scan code for the key, if a key was pressed first, or the pseudo-scan code for the mouse button, if a button was pressed first. (It also returns, via the global variables MouseKeyWordX and MouseKeyWordY, the location of the mouse cursor at the time the key was pressed.) If you have a program that currently uses ReadKeyWord to obtain keyboard input, you can thus add some degree of mouse support just by replacing the call to ReadKeyWord with one to ReadKeyOrButton, and adding some code to handle the button presses.

There are other routines in TPMOUSE that return information about the mouse buttons, but in our experience these are the most useful. Why? Because the other routines can only tell you what buttons are being pressed at any given moment. They don't guarantee that those buttons have also been released. MouseKeyWord and ReadKeyOrButton do, provided that the global variable WaitForButtonRelease is set to True. To see the importance of waiting for the button release, try running some experiments using ReadKeyOrButton with WaitForButtonRelease set to False. Only rarely will a program need WaitForButtonRelease to be set to False. (PREF is exceptional in this respect.)

Mouse Windows and Mouse Coordinates

Also in keeping with the TPCRT model, TPMOUSE allows you to set up "mouse windows" (regions of the screen whose boundaries the mouse cursor must observe), and to read the current window-relative position of the mouse cursor.

To set up a mouse window, you call either of two routines: MouseWindow or FullMouseWindow. MouseWindow is just like Window. It takes four 1-based values as parameters--XLow, YLow, XHigh, YHigh--so to create an 80x25 window, you would use 'MouseWindow(1, 1, 80, 25);'. FullMouseWindow is a "shell routine": all it does is call MouseWindow with the current dimensions of the screen as parameters. Since TPMOUSE calls this procedure for you when your program starts, you usually don't need to call either routine yourself unless you want to create a smaller window, or you have changed video modes or switched in or out of the 8x8 font.

As mentioned in the section on "Reading the Mouse Buttons," you can generally determine the location of the mouse cursor at the time a mouse button was pressed by checking the global variables MouseKeyWordX and MouseKeyWordY. If you

need to know its location at some other time, you have several options. One option is to use MouseWhereX and MouseWhereY, which are just like WhereX and WhereY. A second, and sometimes better one, is to use MouseWhereXY, which returns both the X and Y coordinates of the cursor and the status of the mouse buttons.

In each case, it is assumed that you want to know the position of the mouse cursor *relative to the current mouse window*. If that's not the case, you can easily calculate the absolute position by adding MouseXLo to the X (or column) coordinate, and MouseYLo to the Y (or row) coordinate. Along with MouseXHi and MouseYHi, these variables hold the current dimensions of the mouse window. (MouseXLo and MouseYLo are 0-based numbers, meaning that the top left corner of the screen is 0,0; MouseXHi and MouseYHi are 1-based.)

Mouse Droppings

Now for some bad news. Mouse programming can be extremely frustrating at times, because mice are not tidy animals, at least not when you are using a software cursor. Here's a typical scenario for you to consider:

The mouse cursor is being moved to row 1, column 1. Just before moving it there, the mouse driver looks at the character-attribute pair at that location and saves a copy of it. When the mouse gets moved again, the mouse driver will then restore the pair before moving the cursor to its destination. As long as the contents of the screen don't change, this works fine, but what if your program writes something at row 1, column 1 while the mouse cursor is there? The mouse cursor would get overwritten, of course. That's not so bad, however, because the mouse cursor will be revealed again the next time it is moved. What *is* bad is that the mouse driver will then restore the character and attribute that *used to be* at 1,1. The result is an out-of-place character or oddball color on the screen, and if you aren't careful the screen can get littered with them. Because the effect can resemble a sidewalk attacked by a band of colorful pigeons, these are aptly called "mouse droppings".

So, how do you prevent them from cluttering your carefully designed screens? Unfortunately, the only sure-fire method is to remove the mouse cursor from the screen *every* time you do something to it. If the screen is about to be cleared, scrolled, painted, written to, saved or restored, you really must call HideMouse beforehand and ShowMouse afterward. This is why we said earlier that you will want to use a hardware mouse cursor whenever possible.

Although there's nothing TPMOUSE can do to help you in your battles against mouse droppings, we can at least give you a tip that may prove useful in some situations. If you are doing a lot of screen I/O using a single routine, such as FastWrite, you might want to use a shell routine like this one:

```
procedure MouseWrite(S : String; Row, Col, Attr : Byte);
var
  SaveMouseCursor : Boolean;
begin
  SaveMouseCursor := MouseInstalled and MouseCursorOn;
  if SaveMouseCursor then
    HideMouse;
  FastWrite(S, Row, Col, Attr);
  if SaveMouseCursor then
    ShowMouse;
end;
```

By using a routine like this, you can greatly reduce the number of changes that you have to make to your program. One global search-and-replace operation in your editor can take care of all the FastWrite operations in an entire unit. The price to be paid here is speed: screen writes take longer, and much time can be wasted turning the mouse cursor on and off, especially if you have a lot of screen writing to do. In cases where you have, for example, 6 successive calls to FastWrite, you will therefore probably want to just turn the mouse cursor off before the first one, and turn it on again after the last one.

Mouse Event Handlers

Although the high-level facilities that we've been discussing should be adequate for most applications, there may be times when you need information about what the mouse is doing as soon as an event takes place. (An "event" occurs when one of three things happen: the mouse is moved, a mouse button is pressed, or a mouse button is released.) To get this information, you need to write what is called a "mouse event handler."

It sounds complicated, but TPMOUSE makes the task quite easy. All you have to do is write a FAR-declared global procedure that takes no parameters, and call SetMouseEventHandler to tell TPMOUSE that you want it to be called when a mouse event occurs:

```
{$F+}
procedure MyEventHandler;
begin
  {handle event}
end;
...
SetMouseEventHandler(AllMouseEvents, @MyEventHandler);
```

TPMOUSE will then install its own low-level event handler, which takes the raw data provided by the mouse, massages it into usable form, stores it in global variables, then calls your routine.

SetMouseEventHandler takes two parameters. The second, UserRoutine, is the address of your event handler. The first, EventMask, is a bit-mapped byte that

describes the events that you are interested in. You can construct this byte value using one or more of the following constants:

```
DisableEventHandler  = $00;
MouseMoved           = $01;
LeftButtonPressed    = $02;
LeftButtonReleased   = $04;
RightButtonPressed   = $08;
RightButtonReleased  = $10;
CenterButtonPressed  = $20; {three-button mice only}
CenterButtonReleased = $40; {three-button mice only}
AllMouseEvents       = $7F;
```

For example, if you were interested only in the left mouse button, you would specify 'LeftButtonPressed+LeftButtonReleased' as your EventMask. If you were interested in all events, you would use 'AllMouseEvents'. And if you wanted to disable your event handler at some point, you would use 'DisableEventHandler' (or pass 'nil' as the UserRoutine argument).

Once your event handler is installed, you can obtain all the information you need about the condition of the mouse by checking the following global typed constants:

```
MouseEvent  : MouseEventType = DisableEventHandler;
MouseStatus : ButtonStatus = NoButton;
MouseLastX  : Byte = 1;
MouseLastY  : Byte = 1;
```

MouseEvent is a bit-mapped byte of the same type as EventMask. You can check for a particular event, a mouse movement for example, by ANDing MouseEvent with the appropriate constant and checking for a non-zero result:

```
MouseWasMoved := MouseEvent and MouseMoved <> 0;
```

MouseStatus tells you which mouse buttons, if any, are being pressed. It is declared to be of the enumerated type ButtonStatus:

```
type
  ButtonStatus = (
    NoButton, LeftButton, RightButton, BothButtons,
    {the following values are possible only on a 3-button mouse}
    CenterButton, LeftAndCenterButtons, RightAndCenterButtons, All3Buttons);
```

And you can determine the window-relative location of the mouse cursor by looking at MouseLastX and MouseLastY.

When writing your event handler, keep in mind that events can take place many times a second. It needs to be short and sweet; it can't use much stack space; and it certainly can't call any procedures or functions that are not reentrant: that includes

most of DOS, portions of the Turbo Pascal runtime library, and any routines of your own (or ours) that could be thrown off if the value of a global variable were to change unexpectedly in the middle of the routine.

Using the Mouse in a TSR

Writing a TSR that includes mouse support is not too difficult as long as you keep in mind a few key points. We'll spell these out in a moment, but we should probably warn you that this discussion assumes a working knowledge of how to write a TSR using the TPTSR unit. If you don't have it, you should be sure to at least study the documentation for TPTSR before reading on. Better still, come back to this section after you've written at least one TSR that you're ready to add mouse support to. If you're already comfortable with TSRs, though, by all means read on.

The first point to bear in mind is that your TSR must be prepared to pop up over a program that is already using the mouse. To avoid disturbing it, you must be prepared to save the state of the mouse driver when you pop up, and restore it on exit. The routines you need to do this are MouseStateBufferSize, SaveMouseState, and RestoreMouseState.

MouseStateBufferSize is a function that returns the size of the buffer needed to store all the mouse driver's working data. We strongly recommend that you call it near the start of your program and allocate the buffer before going resident, for reasons that should be obvious to you in time. SaveMouseState and RestoreMouseState do exactly what their names suggest.

The second thing to keep in mind is that you cannot set up the mouse the way you want it before going resident and expect it to be configured the way you want it when you pop up. The program you pop up over might have chosen a different kind of cursor or set up a tiny mouse window, for example. So you have to reinitialize the mouse each time you pop up.

Finally, if you need to use either ReadKeyOrButton, MouseKeyPressed, or MouseKeyWord, you will need to call EnableEventHandling on entry to your popup and DisableEventHandling on exit. The first call is required in order to use the three I/O routines; the second is needed to insure that EnableEventHandling will actually do something the next time you pop up.

Now let's put some of these pieces together. The following is a sketchy outline of a TSR that uses TPMOUSE:

```
program MyTSR;
uses
  Dos, TpCrt, TpMouse, TpInt, TpTsr;
var
```

```
  MSP : MouseStatePtr;
  MSPsize : Word;
  {$F+}
  procedure PopupEntryPoint(var Regs : Registers);
  begin
    ReInitCrt;                          {reinitialize CRT}
    {** save screen and cursor **}
    if MouseInstalled then begin
      SaveMouseState(MSP, False);       {save the state of the mouse driver}
      InitializeMouse;                  {reinitialize the mouse}
      EnableEventHandling;              {install event handler}
      {** set up a mouse window **}
      {** choose a mouse cursor and make it visible **}
    end;
    {** body of the popup goes here **}
    {** restore screen and cursor **}
    if MouseInstalled then begin
      DisableEventHandling;             {disable event handler}
      RestoreMouseState(MSP, False);    {restore state of the mouse driver}
    end;
  end;
  {$F-}

begin
  {...}
  if MouseInstalled then begin
    MSPsize := MouseStateBufferSize;    {allocate mouse state buffer}
    if MSPsize = 0 then                 {check the size}
      MouseInstalled := False
    else
      GetMem(MSP, MSPsize);             {allocate the buffer}
  end;
  {...}
end.
```

The only additional points worth making here are that you should be sure to 1) call ReinitCrt before saving the screen, 2) save the state of the mouse driver *after* saving the screen, and 3) restore the state of the mouse *after* restoring the screen. If you don't take these steps in the right order, you can easily corrupt the screen of the program you pop up over. If you need a more complete model than MyTSR, study the demonstration program PREF.

Inequality among Mouse Drivers

Despite what was just said in the last paragraph, there is one more *very important* thing you need to know about MouseStateBufferSize, SaveMouseState, and RestoreMouseState. The function calls that they use aren't implemented by all mouse drivers. MicroSoft didn't have any need for them until they wrote CodeView, which had to be capable of debugging programs that used the mouse, so in the early versions of the MicroSoft mouse driver these functions didn't exist. They are probably implemented in the current versions of all mouse vendors' device drivers, but of course not everyone has the current versions.

The hard truth is that you have to be careful about using these functions (GetMousePage and SetMousePage fall into the same category) in your programs. We have attempted to make MouseStateBufferSize return 0 when it encounters a mouse driver that doesn't implement them, but don't count on it. You should either provide your users with a means of disabling mouse support manually, or else issue stern warnings advising them that your program isn't guaranteed to work correctly with older mouse drivers.

The Mickey

One last bit of technical data. The smallest unit of distance recognized by a mouse is a "mickey," roughly 1/100 of an inch. As you move a mouse across your desk, the mouse hardware must pass back to the mouse driver information that indicates how far the mouse has been moved, and this data is expressed in terms of mickeys. The mouse driver, in turn, must then determine if and how far the mouse cursor needs to be moved on the screen to reflect the changes in the position of the mouse. In making these determinations, it relies on a value known as the "mickey-to-pixel ratio".

The mickey-to-pixel ratio indicates the number of mickeys that the mouse must be moved in order to move the mouse cursor eight pixels. Actually, however, there are two such ratios: one pertaining to horizontal movement, and one to vertical movement. The default ratios are 8 and 16, respectively. In 80-column text modes, the default settings require the mouse to be moved about 8/100 of an inch horizontally in order to move the mouse cursor one column to the left or right, and 16/100 of an inch up or down in order to move it by one column.

Confused? Fortunately, you don't need to understand much about the mechanics of mice in order to support them in your software. This section is provided primarily as background material for the discussions of the SetMickeyToPixelRatio and GetMickeyCount routines, described below, which most people will never need to use. (See the demonstration program MAKEMENU for an example of a case where SetMickeyToPixelRatio is useful.)

TPMOUSE: Declarations

Types

```
ButtonStatus = (
  NoButton, LeftButton, RightButton, BothButtons,
  CenterButton, LeftAndCenterButtons, RightAndCenterButtons, All3Buttons);
```

Used in routines that check to see whether a particular mouse button, or combination of buttons, has been pressed. On a two-button mouse, only the values NoButton..BothButtons are meaningful.

```
MouseEventType = DisableEventHandler..AllMouseEvents;
```

A mouse event (mouse was moved, button was pressed, button was released).

```
MouseState =
  record
    BufSize : Word;
    Buffer  : array[1..400] of Byte;
  end;
MouseStatePtr = ^MouseState;
```

Data structure used to store the state of the mouse driver. Buffer is a variably-sized array. BufSize is equal to the size of the array plus 2.

```
MouseXLo : Byte;
MouseXHi : Byte;
MouseYLo : Byte;
MouseYHi : Byte;
```

Screen coordinates for the current mouse window. As with the WindMin and WindMax variables in CRT and TPCRT, MouseXLo and MouseYLo are 0-based numbers, while MouseXHi and MouseYHi are 1-based numbers.

Constants

```
DefaultScreenMask = $FFFF;
DefaultCursorMask = $7700;
```

The default screen and cursor masks for a software mouse cursor.

```
DisableEventHandler  = $00;
MouseMoved           = $01;
LeftButtonPressed    = $02;
LeftButtonReleased   = $04;
RightButtonPressed   = $08;
RightButtonReleased  = $10;
CenterButtonPressed  = $20;
CenterButtonReleased = $40;
AllMouseEvents       = $7F;
```

Bit masks for the various mouse events.

```
MouseEvent  : MouseEventType = DisableEventHandler;
MouseStatus : ButtonStatus = NoButton;
MouseLastX  : Byte = 1;
MouseLastY  : Byte = 1;
```

Used to pass information to event handlers that have been set up using SetMouseEventHandler.

```
MouseLft     = $EF00; {left button}
MouseRt      = $EE00; {right button}
MouseBoth    = $ED00; {both buttons}
MouseCtr     = $EC00; {center button}
MouseLftCtr  = $EB00; {left and center buttons}
MouseRtCtr   = $EA00; {right and center buttons}
MouseThree   = $E900; {all three buttons}
```

These are the pseudo-scan codes returned by MouseKeyWord and ReadKeyOrButton when a mouse button (or combination of buttons) has been pressed.

```
MouseRoutine : Pointer = nil;
```

If not nil, MouseRoutine points to a user-written event handler set up using SetMouseEventHandler.

```
MouseRoutineEvent : MouseEventType = DisableEventHandler;
```

Bit-mapped byte indicating the event(s) that will cause the procedure pointed to by MouseRoutine to be called.

```
WaitForButtonRelease : Boolean = True;
```

If True, MouseKeyWord waits for the mouse button currently being pressed to be released before returning a pseudo-scan code.

Variables

```
MouseCursorOn : Boolean;
```

This variable keeps track of whether the mouse cursor is currently being displayed or not.

```
MouseInstalled : Boolean;
```

Indicates whether a mouse is installed or not.

```
MouseKeyWordX : Byte;
MouseKeyWordY : Byte;
```

These hold the coordinates of the mouse cursor at the time of the last call to MouseKeyWord.

BlockMouseCursor / DisableEventHandling

Declaration

```
procedure BlockMouseCursor;
```

Purpose

Set scan lines for a block mouse cursor.

Comments

This routine selects a hardware mouse cursor shaped like a large block. It is analogous to the BlockCursor routine in TPCRT.

See Also

BlockCursor (TPCRT)

HardMouseCursor

Declaration

```
procedure DisableEventHandling;
```

Purpose

Disable the event handler installed by EnableEventHandling.

Comments

This routine disables TPMOUSE's internal event handler, which is installed when EnableEventHandling is called. Its principal use is in TSRs that need to use MousePressed and MouseKeyWord. See the entry for EnableEventHandling for an example.

See Also

EnableEventHandling

SetMouseEventHandler

EnableEventHandling / FatMouseCursor

Declaration

```
procedure EnableEventHandling;
```

Purpose

Enable the event handler needed for MousePressed and MouseKeyWord.

Comments

This routine must be called before MousePressed or MouseKeyWord may be used in a program. What it does is to activate TPMOUSE's mouse event handler, a routine which will then be called by the mouse driver each time that the mouse cursor is moved or a mouse button is pressed or released.

For more details, see the discussions on "Reading The Mouse Buttons" and "Mouse Event Handlers" at the beginning of this section.

Example

```
EnableEventHandling;
...
if MousePressed then begin
  PseudoKey := MouseKeyWord;
  {handle command assigned to PseudoKey}
end;
```

Activate the event handler, then use MousePressed and MouseKeyWord to obtain mouse button information

See Also

DisableEventHandling
MousePressed
MouseKeyWord
SetMouseEventHandler

Declaration

```
procedure FatMouseCursor;
```

Purpose

Set larger scan lines for mouse cursor based on current video mode.

Comments

This routine selects a hardware mouse cursor that is slightly larger than a normal two-line cursor. It is analogous to the FatCursor routine in TPCRT.

See Also

FatCursor (TPCRT)
HardMouseCursor

FullMouseWindow / GetMickeyCount

Declaration

```
procedure FullMouseWindow;
```

Purpose

Sets mouse window coordinates to full screen.

Comments

This routine selects the entire screen as the current mouse window. Note that there is no need to explicitly call this routine at the beginning of a program, since it will already have been called if a mouse is installed.

Example

```
SetFont8x8(True);
FullMouseWindow;
```

Switch into 43-/50-line mode and reset the mouse's window coordinates.

See Also

MouseWindow

Declaration

```
procedure GetMickeyCount(var Horizontal, Vertical : Integer);
```

Purpose

Return the horizontal and vertical mickey count since the last call to this function.

Comments

This routine returns values indicating how far the mouse has been moved since the last time GetMickeyCount was called. Negative numbers indicate movement up (Horizontal) or to the left (Vertical); positive numbers indicate movement down (Horizontal) or to the right (Vertical).

GetMickeyCount is perhaps most useful in cases where you need to be able to determine whether or not the user has been moving the mouse, even if the reported position of the mouse cursor remains unchanged because it has reached the edge of the mouse window. For example, if the mouse cursor is located at row 1, column 1, and GetMickeyCount returns a Horizontal value of 0 and a Vertical value of -16, then you would know that user had been trying to move the mouse upward.

See the discussion of "The Mickey" at the beginning of this section for an explanation of the term "mickey".

See Also

MouseWhereXY

SetMickeyToPixelRatio

Declaration

```
function GetMousePage : Byte;
```

Purpose

Return the video page where the mouse is being displayed.

Comments

This routine returns the number of the video page where the mouse cursor is currently being displayed.

Warning: This routine uses a mouse function that is not implemented in all mouse drivers. See the discussion of MouseStateBufferSize for more details.

Example

```
SaveMousePage := GetMousePage;
SetMousePage(NewMousePage);
...
SetMousePage(SaveMousePage);
```

This saves the current mouse page, switches to a new page, then restores the old one.

See Also

SetMousePage

HardMouseCursor

Declaration

```
procedure HardMouseCursor(StartLine, EndLine : Word);
```

Purpose

Set mouse to use the hardware cursor.

Comments

This routine does two things: it selects a hardware mouse cursor rather than a software cursor (the mouse driver's default choice), and it lets you specify the starting (StartLine) and ending (EndLine) scan lines for that cursor. This is a low-level routine, and in most cases you will probably prefer to use one of the higher level routines in TPMOUSE that call it: BlockMouseCursor, FatMouseCursor, HiddenMouseCursor, or NormalMouseCursor.

For more details, see the discussion of "The Mouse Cursor" at the beginning of this section.

Example

```
if Font8x8Selected or (CurrentMode <> 7) then
  HardMouseCursor(0, 7)
else
  HardMouseCursor(0, 12);
```

Like the BlockMouseCursor routine, this would select a hardware cursor shaped like a solid block.

See Also

BlockMouseCursor
FatMouseCursor
HiddenMouseCursor
NormalMouseCursor
SoftMouseCursor

Declaration

```
procedure HiddenMouseCursor;
```

Purpose

Hide the mouse cursor.

Comments

This routine selects a hardware mouse cursor that is (at least temporarily) hidden. It is analogous to the HiddenCursor routine in TPCRT. Note that this routine does not perform the same function as HideMouse, which can hide either a software or a hardware mouse cursor.

See Also

HardMouseCursor

HiddenCursor (TPCRT)

Declaration

```
procedure HideMouse;
```

Purpose

Hide the mouse cursor.

Comments

This routine hides the mouse cursor, no matter what its kind or shape. Probably the most common use for it is in situations where a software cursor is being used, and either the screen is about to be scrolled or a copy of the screen is about to be made (by a routine in TPWINDOW, perhaps).

It is not necessary to check MouseInstalled before calling HideMouse, because HideMouse does nothing if MouseInstalled is False.

For more details, see the discussion of "The Mouse Cursor" at the beginning of this section.

Example

```
HideMouse;
{write to the screen, scroll display, save an image of the display, whatever}
ShowMouse;
```

Hide the mouse cursor, do something to the display, then reveal the mouse cursor.

See Also

InitializeMouse

ShowMouse

InitializeMouse

Declaration

```
procedure InitializeMouse;
```

Purpose

Reinitializes mouse and sets MouseInstalled.

Comments

This routine reinitializes the mouse and sets MouseInstalled. Although it is called automatically at the beginning of the program, there are cases where it might need to be called later as well--in a TSR for example. In "reinitializing" the mouse, it does the following:

1) Hides the mouse cursor. 2) Selects a software mouse cursor (still hidden) with a screen mask of $FFFF (DefaultScreenMask) and a cursor mask of $7700 (DefaultCursorMask). 3) Selects an 80 x 25 mouse window (40 x 25 in 40-column modes). 4) Moves the cursor to the center of that window. 5) Selects a mickey-to-pixel ratio of 8,16. And 6) disables any user-defined event handler.

Example

```
SaveMouseState(MSP, False);
InitializeMouse;
EnableEventHandling;
BlockMouseCursor;
ShowMouse;
...
DisableEventHandling;
RestoreMouseState(MSP, False);
```

A fairly typical series of statements in a TSR that uses the mouse: Save the state of the mouse driver, reinitialize the mouse, then select a hardware mouse cursor and make it visible. Later the previous state of the mouse driver is restored.

See Also

HideMouse

ShowMouse

MouseButtonPressed / MouseButtonReleased

Declaration

```
function MouseButtonPressed(Button : ButtonStatus; var Count : Word;
                            var LastX, LastY : Byte) : Boolean;
```

Purpose

Return True if the Button to check has been pressed.

Comments

This routine allows you to determine whether or not a particular mouse Button has been pressed. If it has, Count returns the number of times it has been pressed since the last call to MouseButtonPressed, and LastX and LastY will contain the mouse cursor's position the last time it was pressed.

In general, we recommend that you use an event handler in cases where you are tempted to use this function and its companion, MouseButtonReleased. These two routines are provided primarily for completeness.

Example

```
if MouseButtonPressed(LeftButton, Count, LastX, LastY) then
  WriteLn('The left mouse button has been pressed ', Count, ' times);
```

Display the number of times that the left mouse button has been pressed.

See Also

MouseWhereXY

Declaration

```
function MouseButtonReleased(Button : ButtonStatus; var Count : Word;
                             var LastX, LastY : Byte) : Boolean;
```

Purpose

Return True if the Button to check has been released.

Comments

This routine allows you to determine whether or not a particular mouse Button has been released. If it has, Count returns the number of times it has been released since the last call to MouseButtonReleased, and LastX and LastY will contain the mouse cursor's position the last time it was released.

Example

```
if MouseButtonReleased(LeftButton, Count, LastX, LastY) then
  WriteLn('The left mouse button has been released ', Count, ' times);
```

Display the number of times that the left mouse button has been released.

MouseGotoXY / MouseInWindow

Declaration

```
procedure MouseGotoXY(MouseX, MouseY : Byte);
```

Purpose

Set mouse position.

Comments

This routine is analogous to the GotoXY routine in CRT and TPCRT. It moves the mouse cursor to the specified location. As elsewhere in TPMOUSE, the screen coordinates specified here are interpreted as being relative to the current mouse window. If the coordinates are outside the mouse window, MouseGotoXY does nothing.

Example

```
MouseWindow(5, 5, 75, 20);
MouseGotoXY(1, 1);
```

Moves the mouse cursor to row 5, column 5.

See Also

MouseWindow

Declaration

```
function MouseInWindow(XLo, YLo, XHi, YHi : Byte) : Boolean;
```

Purpose

Return True if the mouse cursor is within the specified window.

Comments

This routine returns True if the mouse cursor is currently within the specified region of the screen. Note that the screen coordinates passed in as parameters are interpreted as being relative to the current mouse window.

Example

```
MouseWindow(5, 5, 75, 20);
...
if MouseInWindow(1, 1, 3, 3) then {do something};
```

This checks to see if the mouse cursor is somewhere in the first three rows and first three columns of the current mouse window.

See Also

MouseWindow

Declaration

```
function MouseKeyWord : Word;
```

Purpose

Return a pseudo-scan code based on which mouse button is being pressed.

Comments

This routine is analogous to the ReadKeyWord function in TPCRT. Assuming that EnableEventHandling has been called, it sits in a loop waiting for a mouse button to be pressed, then returns a pseudo-scan code in the range MouseLft..MouseThree that indicates which button or buttons were pressed. If EnableEventHandling has not been called, it will return $FFFF.

Once a mouse button has been pressed, MouseKeyWord does one of two things, depending on the current value of WaitForButtonRelease. If this variable is False, it will immediately return the pseudo-scan code for the button or combination of buttons that was being pressed. If it is True, MouseKeyWord will continue to monitor the status of the mouse buttons until they have all been released. In either case, on exit the routine will store the current location of the mouse cursor in MouseKeyWordX and MouseKeyWordY.

See the discussion on "Reading The Mouse Buttons" at the beginning of this section for more details.

Example

```
ScanCode := $FFFF;
repeat
  if KeyPressed then
    ScanCode := ReadKeyWord
  else if MousePressed then
    ScanCode := MouseKeyWord;
until ScanCode <> $FFFF;
```

Loop until a key or a mouse button is pressed. (This is essentially what ReadKeyOrButton does.)

See Also

EnableEventHandling
MouseKeyWord
MouseWhereXY
ReadKeyOrButton

MousePressed / MouseStateBufferSize

Declaration

```
function MousePressed : Boolean;
```

Purpose

Return True if a mouse button is currently being pressed.

Comments

This routine is analogous to the KeyPressed function in CRT and TPCRT. It tells you whether or not a mouse button is currently being pressed. Like MouseKeyWord, it does nothing useful if EnableEventHandling has not been called. (That is, it always returns False.) See the entry for MouseKeyWord for an example.

See Also

EnableEventHandling MouseKeyWord

Declaration

```
function MouseStateBufferSize : Word;
```

Purpose

Return amount of memory needed to save the state of the mouse driver.

Comments

This function returns the size of the buffer needed to save the state of the mouse driver. It is intended primarily for use in TSRs that need to provide mouse support. The value returned will vary from one mouse driver to another, but should be roughly in the range 350-400 (bytes). If a mouse is not installed, 0 will be returned.

Warning: This routine uses a mouse function that is not implemented in all mouse drivers. It is written in such a way that 0 should be returned if the function isn't implemented, but there's no guarantee that that will happen. You should therefore (1) check for a return value of 0 and (2) warn users of the potential problem. If a problem does arise, they can probably correct it only by updating their mouse driver software.

For an example of how to use this routine, see the introductory section on "Using the Mouse in a TSR."

See Also

RestoreMouseState SaveMouseState

MouseWhereX / MouseWhereXY

Declaration

```
function MouseWhereX : Byte;
```

Purpose

Return current X coordinate for mouse.

Comments

This routine is analogous to the WhereX function in CRT and TPCRT. It returns the present X (column) coordinate for the mouse cursor, relative to the current mouse window. To obtain the absolute column coordinate of the mouse, add MouseXLo to the value returned by MouseWhereX.

See Also

MouseWhereY MouseWindow

Declaration

```
procedure MouseWhereXY(var MouseX, MouseY : Byte; var Status : ButtonStatus);
```

Purpose

Return mouse position and button status.

Comments

This routine returns the present location of the mouse cursor, relative to the current mouse window, in MouseX and MouseY. (To obtain the absolute row and column coordinates of the mouse, add MouseXLo and MouseYLo to the values returned in MouseX and MouseY.) The current state of the mouse buttons is returned in Status. This routine is especially useful in cases where you need to determine which buttons, if any, are being pressed, and execute different functions based on the location of the mouse cursor.

Example

```
MouseWhereXY(MouseX, MouseY, Status);
case Status of
  LeftButton  : HandleLeftButton;  {left button pressed}
  RightButton : HandleRightButton; {right button pressed}
  BothButtons : HandleBothButtons; {left and right buttons both pressed}
end;
```

Get the location of the mouse cursor and the status of the mouse buttons, then execute a particular procedure based on which combination of buttons has been pressed.

See Also

MouseWhereY MouseWindow

MouseWhereY / MouseWindow

Declaration

```
function MouseWhereY : Byte;
```

Purpose

Return current Y coordinate for mouse.

Comments

This routine is analogous to the WhereY function in CRT and TPCRT. It returns the present Y (row) coordinate for the mouse cursor, relative to the current mouse window. To obtain the absolute row coordinate of the mouse, add MouseYLo to the value returned by MouseWhereY.

See Also

MouseWhereX MouseWhereXY

Declaration

```
procedure MouseWindow(XLow, YLow, XHigh, YHigh : Byte);
```

Purpose

Set window coordinates to be observed by the mouse.

Comments

This routine is similar to the Window procedure in CRT and TPCRT. It sets the minimum/maximum horizontal and vertical positions on the screen for the mouse cursor. The screen coordinates are subjected to the same kind of scrutiny as those passed to Window: XLow/YLow cannot be greater than XHigh/YHigh, XHigh cannot be greater than ScreenWidth, and YHigh cannot be greater than ScreenHeight. If any of the coordinates is invalid, MouseWindow does nothing. If the coordinates are valid, however, the mouse cursor will be moved inside the window if isn't already, and the new window coordinates will be stored in MouseXLo, MouseXHi, MouseYLo, MouseYHi.

The default mouse window is the entire screen, based on the width and height of the screen when the program began. If the dimensions of the screen change later on, you should call FullMouseWindow to reset the mouse's window coordinates.

Example

```
Window(1, 1, 80, 25);
if MouseInstalled then
  MouseWindow(1, 1, 80, 25);
```

Create an 80 x 25 window and a mouse window to match. Notice that MouseWindow is not called unless MouseInstalled is True.

See Also

FullMouseWindow

NormalMouseCursor / ReadKeyOrButton

Declaration

```
procedure NormalMouseCursor;
```

Purpose

Set normal scan lines for mouse cursor based on current video mode.

Comments

This routine selects a hardware mouse cursor that is two-lines high. It is analogous to the NormalCursor routine in TPCRT.

See Also

HardMouseCursor

Declaration

```
function ReadKeyOrButton : Word;
```

Purpose

Return code for next key or mouse button pressed.

Comments

This routine is provided partly for internal use, and partly as a convenience. It simply sits in a loop waiting for a key or mouse button to be pressed, then returns either the scan code for the key, if a key was pressed first, or the pseudo-scan code for the mouse button, if a button was pressed first. Assuming that a mouse button was pressed first, on exit the current location of the mouse cursor will be stored in MouseKeyWordX and MouseKeyWordY.

ReadKeyOrButton uses MousePressed and MouseKeyWord to obtain information about the mouse buttons, so it is subject to the same limitation that they are: EnableEventHandling must be called before it can obtain any information about the mouse buttons. See the discussion on "Reading The Mouse Buttons" at the beginning of this section for more details.

Example

```
I := Integer(ReadKeyOrButton);
if Lo(I) = 0 then
  case I of
    $0000 : {^Break was pressed} ;
    ...
    Integer(MouseLft) : {left mouse button was pressed} ;
  end;
```

Get the next key or mouse button press and process the command. Notice that MouseLft must be typecast to an integer, since Turbo Pascal does not allow word values as case labels.

See Also

EnableEventHandling (TPCRT)
ReadKeyWord (TPCRT)
MouseKeyWord (TPCRT)

RestoreMouseState / SaveMouseState

Declaration

```
procedure RestoreMouseState(var MSP : MouseStatePtr; Deallocate : Boolean);
```

Purpose

Restore the state of the mouse driver and Deallocate the buffer if requested.

Comments

This function restores the state of the mouse driver. It is intended primarily for use in TSRs that need to provide mouse support. On entry, MSP points to the buffer containing the data to be restored. If Deallocate is True, this buffer will be deallocated on exit and MSP will be set to nil. If a mouse is not installed or MSP is nil on entry, RestoreMouseState will do nothing.

Warning: This routine uses a mouse function that is not implemented in all mouse drivers. See the discussion of MouseStateBufferSize for more details.

For an example of how to use this routine, see the introductory section on "Using the Mouse in a TSR."

Declaration

```
procedure SaveMouseState(var MSP : MouseStatePtr; Allocate : Boolean);
```

Purpose

Save the state of the mouse driver, allocating the buffer if requested.

Comments

This function saves the state of the mouse driver. It is intended primarily for use in TSRs that need to provide mouse support. MSP points to the buffer used to save the mouse driver's data. If Allocate is True, this buffer will be allocated on the heap and MSP will point to it on exit. If it is False, MSP should point on entry to a previously allocated buffer. If a mouse is not installed, or MouseStateBufferSize returns 0 when asked for the size of the buffer needed, MSP will be set to nil.

Warning: This routine uses a mouse function that is not implemented in all mouse drivers. See the discussion of MouseStateBufferSize for more details.

For an example of how to use this routine, see the introductory section on "Using the Mouse in a TSR."

See Also

MouseStateBufferSize

Declaration

```
procedure SetMickeyToPixelRatio(Horizontal, Vertical : Integer);
```

Purpose

Sets the mickey-to-pixel ratio.

Comments

For an explanation of the concepts of the "mickey" and the "mickey-to-pixel ratio", see the discussion of "The Mickey" at the beginning of this section.

This routine lets you alter the mickey-to-pixel ratio(s). Although the default settings of 8,16 (Horizontal,Vertical) are usually appropriate, there may be cases where you want to increase or decrease the mouse's sensitivity to movement. A setting of 16,32 will "slow down" the mouse considerably by making it less sensitive to movement, while a setting of 4,8 will make it "move faster" by increasing its sensitivity to movement.

The Horizontal and Vertical parameters should both be integers in the range 1 to 32767 (MaxInt).

Examples

```
SetMickeyToPixelRatio(4,8);
```

Speed things up.

```
SetMickeyToPixelRatio(16,32);
```

Slow things down.

SetMouseEventHandler

Declaration

```
procedure SetMouseEventHandler(EventMask : MouseEventType;
                               UserRoutine : Pointer);
```

Purpose

Sets the address of a routine to be called when the specified mouse events occur.

Comments

Unlike the frequently-mentioned EnableEventHandling routine, the main purpose of this routine is not to activate TPMOUSE's own event handler (although that will be done if necessary), but rather to specify a user-written routine (pointed to by UserRoutine) that it is to call when selected mouse events occur.

If not nil, UserRoutine should point to a FAR-declared procedure that takes no parameters. Information about the Event that just took place can be obtained by checking the global variables MouseEvent, MouseStatus, MouseLastX, and MouseLastY. If UserRoutine is nil (or EventMask = DisableEventHandler), then any user-defined event handler currently in place will be "disabled" (not called).

EventMask is a bit-mapped byte that tells TPMOUSE's event handler when your routine should be called. For example, if you want it to be called only when the left button is pressed or released, the EventMask should be 'LeftButtonPressed+ LeftButtonReleased'. If you wanted to know about all mouse events, you would use AllMouseEvents instead.

Note that TPMOUSE handles all the "dirty work" involved in writing an event handler for you: it saves all the CPU registers, sets DS to point to your program's own data segment, calls UserRoutine, then restores the CPU registers when it regains control.

For more details, see the discussions of "Mouse Event Handlers" at the beginning of this section.

See Also

DisableEventHandling EnableEventHandling

Declaration

```
procedure SetMousePage(Page : Byte);
```

Purpose

Sets the video page where the mouse will be displayed.

Comments

This routine, analogous to the SetVisiblePage routine in TPCRT, tells the mouse driver which video page to use when displaying the mouse cursor.

Warning: This routine uses a mouse function that is not implemented in all mouse drivers. See the discussion of MouseStateBufferSize for more details.

Example

```
SetVisiblePage(3);
SetMousePage(CurrentPage);
```

Switch both the display and the mouse cursor to video page 3. Notice that SetMousePage uses CurrentPage, in case the call to SetVisiblePage failed.

See Also

GetMousePage

SetVisiblePage (TPCRT)

Declaration

```
procedure ShowMouse;
```

Purpose

Show the mouse cursor.

Comments

This routine must be called in order to reveal the mouse cursor, whether it is a software cursor or a hardware cursor.

It is not necessary to check MouseInstalled before calling ShowMouse, because ShowMouse does nothing if MouseInstalled is False.

For more details, see the discussion of "The Mouse Cursor" at the beginning of this section.

Example

```
BlockMouseCursor;
ShowMouse;
...
HideMouse;
```

Select a hardware mouse cursor shaped like a block, unhide the cursor, do something, then re-hide the cursor.

See Also

HideMouse

InitMouse

SoftMouseCursor

Declaration

```
procedure SoftMouseCursor(ScreenMask, CursorMask : Word);
```

Purpose

Set mouse to use a software cursor.

Comments

This routine does two things: 1) it selects a software cursor (the default) rather than a hardware cursor, and 2) it lets you specify, in a somewhat roundabout fashion, the shape and color of the cursor.

For details on the subject of mouse cursors, including screen masks and cursor masks, see the discussion of "The Mouse Cursor" at the beginning of this section.

Examples

```
SoftMouseCursor(DefaultScreenMask, DefaultCursorMask);
```

Selects the default type of software cursor, a non-blinking block whose color varies depending on where the cursor is located.

```
SoftMouseCursor($00FF, $F000);
```

Selects a blinking reverse video (black on white) block cursor.

```
SoftMouseCursor($0000, $1F04);
```

Selects a white diamond (ASCII 4) on a blue background for a cursor.

See Also

HardMouseCursor

H. TPPICK: Pick Lists

Whenever a program must prompt for a choice from a list, TPPICK can perform the necessary low-level actions. Whether the list contains a directory of files, a set of configuration options, or a collection of states and zip codes, TPPICK provides the necessary flexibility to display the list, let the user browse through it, and select an entry.

TPPICK and TPMENU provide similar but complementary capabilities. TPMENU can select an item from a multi-level hierarchical list, but the list must be fairly static, known in advance. By contrast, TPPICK's list comprises a single level, but the list may contain an arbitrary number of items, and the items may be determined at runtime, even just prior to displaying and picking from the list. TPPICK even may be simpler to use for static single-level menu systems.

The central routine of TPPICK is PickWindow. A single call to this routine opens a window, displays a list of items, accepts keystrokes to browse the list, restores the screen, and returns the chosen item.

TPPICK offers four avenues of flexibility to handle diverse lists. First, the PickWindow routine takes parameters that specify window position, colors, and the source of the item list. Second, your program generates the item list by providing a function to associate an item index with a string to be displayed. Third, the interface section of TPPICK declares a number of typed constants that your program can adjust to affect various facets of the "pick list" display: how many columns of items, the minimum and maximum number of screen rows to use, whether to allow searching for items within the list, and so on. Like many other Turbo Professional units, TPPICK has a redefinable keyboard interface--you can change the keystrokes, add or delete commands, and modify the source of keyboard input using the methods described in Appendix B. TPPICK also supports the mouse, including the use of a scroll bar for quick access to other parts of a long list.

The following short example demonstrates the use of TPPICK. Here, the user selects from a list of five pizza toppings. Selections are made by using the cursor keys or by pressing the first letter of any item name.

```
program PickExample;
uses
  TpCrt, TpString, TpWindow, TpCmd, TpPick;
const
  NumPizzaToppings = 5; {Number of items on pick list}
  PickColors : PickColorArray =
    ($07,                {Unselected item color}
     $0F,                {Window frame color}
     $70,                {Window title color}
```

```
      $70,                {Selected item color}
      $07,                {Alternate unselected item color}
      $0F                 {Alternate selected item color}
     );
var
  Choice : Word;          {Which item was chosen}

{$F+}                     {Must be FAR}
function PizzaTopping(Item : Word) : string;
  {-Return the name of each item}
begin
  case Item of
    1 : PizzaTopping := ' Pepperoni';
    2 : PizzaTopping := ' Sausage';
    3 : PizzaTopping := ' Mushrooms';
    4 : PizzaTopping := ' Anchovies';
    5 : PizzaTopping := ' Onions';
  end;
end;
{$F-}

begin
  ClrScr;
  WriteLn('Select a pizza topping');

  TpPick.PickSrch := CharPickNow;      {Accept first letter name selections}
  Choice := 1;                         {Set initial choice}

  if not PickWindow(@PizzaTopping,     {Routine that returns item names}
                    NumPizzaToppings,  {Number of items in pick list}
                    5, 5, 17, 11,      {Coordinates of the window}
                    True,              {Draw a frame around the list}
                    PickColors,        {Colors to use}
                    '',                {No title for this window}
                    Choice)            {Which item was chosen}
  then
    {Insufficient memory for screen buffer}
    WriteLn('Unable to execute PickWindow')
  else if PickCmdNum = PKSSelect then
    {A selection was made}
    WriteLn('You chose:', PizzaTopping(Choice))
  else
    {Escape was pressed}
    WriteLn('No choice was made');
end.
```

The pick routines refer to as many as six separate video attributes in order to display the list. The constant PickColors here defines those colors. You may wish to have two such constants, one used on a color monitor and another for monochrome.

The example function PizzaTopping is important to understand. Each time you use the pick routines, you must define a function that matches this prototype: a global function compiled under the FAR model that takes a single Word parameter and returns a String. You pass the address of your routine as a parameter to

PickWindow. Each time the pick routines need to get the name of an item on the pick list, they call your routine. The parameter passed to the routine will always be in the range 1..NumItems, where NumItems is another parameter that you pass to PickWindow. The string you return may be determined any way you like: by using a case statement as shown here, looking up the name in an array of strings, or computing it based on other variables. PickWindow will not add leading spaces to the string you return, but it will pad or trim the string to the required width.

PickWindow supports several search options. By default, no searching is performed. That is, the only way to move the pick highlight bar is by pressing the cursor keys. After selecting other TPPICK options, the pick routines will move the highlight bar when the user enters alphanumeric characters matching the names of menu items. In the example, we set the PickSrch constant declared in the TPPICK unit to CharPickNow. This means that when an alphanumeric character matching the first letter of a pick item is entered, the pick routine will immediately exit, returning the index of the next matching item. This behavior matches that of TPMENU.

In the example, we designate which item to highlight initially by assigning a value of 1 to Choice.

Next, the example calls the PickWindow routine. Parameters to this call include the address of the string function, PizzaTopping; the number of items on this list; the upper left and lower right coordinates of the pick window; whether to frame the window or not; the color values to use; a title to display on the window border; and finally a VAR parameter that contains the initial choice and the final selection made by the user.

If PickWindow returns False, a severe error occurred. Either the parameters passed to PickWindow were incorrect, or not enough heap space was available to buffer the screen covered by the pick window.

If PickWindow returns True, then the pick window was successfully displayed, and the user performed some action to exit the window. The global variable PickCmdNum interfaced by TPPICK will contain a code indicating which action exited the window. In this example, we only care about the PKSSelect action, which means that the user pressed either <Enter> or an alphanumeric key that matched one of the items. Then we can use the Choice number returned by PickWindow, along with the string function PizzaTopping, to display the name of the selected choice.

This simple example barely scratches the surface of what TPPICK can do. The TPDIR unit is a somewhat more thorough example which uses to TPPICK to perform filename selections.

Special thanks to Dan T. Davis for his contributions to earlier versions of the TPPICK unit.

Pick List Orientations

When the list items are arranged in more than one column, TPPICK can display the items in any of three different orientations. It's easier to demonstrate these orientations visually than to describe them. Assume that you have an 11 item list (the letters A through K) arranged in three columns in a window three rows tall. Here's what you would see for each of the three orientations when the list is first displayed:

```
PickVertical        PickHorizontal        PickSnaking
 A   E   I           A   B   C            A   D   G
 B   F   J           D   E   F            B   E   H
 C   G   K           G   H   I            C   F   I
```

For PickVertical, increasing item numbers are arranged in vertical columns. Note that items D and H are below the window and do not appear until the window scrolls upward.

For PickHorizontal, increasing items are arranged in horizontal rows. The right hand edge of one row is continued immediately at the left edge of the next row. Scrolling the example up by one row would show the two remaining elements, J and K.

PickSnaking is like PickVertical in that the items are arranged in columns. However, it is like PickHorizontal in that the top of one column picks up where the bottom of the previous column left off. Scrolling the example window left by one column makes items J and K appear in the right hand column.

Which orientation is best depends on the application. For sorted directory lists, we prefer the PickSnaking orientation because the eye naturally tends to scan down the columns and yet, unlike PickVertical, no breaks in the sequence occur.

Note that the list orientation has no effect on the use of the cursor keys within the list. Pressing the <Down> arrow key, for example, always moves the highlight bar down one row.

PickVertical is active by default. You can change the orientation by the following steps:

1. Activate the conditional define EnablePickOrientations in TPDEFINE.INC, and recompile TPPICK.PAS.
2. Before you call any other routines from TPPICK, call SetVerticalPick,

SetHorizontalPick, or SetSnakingPick to set the orientation. By calling one of these routines rather than setting a typed constant, you avoid linking in the code from all of the orientations. Be warned that if you forget to call one of these routines, all the other TPPICK procedures and functions will just exit with an error.

Important: Please keep in mind that activating EnablePickOrientations indirectly affects two other units as well: TPDIR and TPHELP. Both of these units use TPPICK (to display directory lists and help indexes, respectively). If you are using one of them you need to call one of the three routines named above, even if your program does not explicitly use TPPICK.

It is possible to use more than one orientation in a given program. Just call one of the three routines prior to calling PickWindow, FillPickWindow, or PickBar. The PickOrientation function is provided to determine which orientation is currently active.

TPPICK is reentrant, so you can safely change orientations for one pick window while another one is already on-screen.

Protected Items

Pick lists can contain items that are never meant to be selected. Such items might be used as subtitles, white space, or separator bars that scroll along with the list itself.

In order to use this capability, you must take several steps:

1. Activate the conditional define PickItemDisable in TPDEFINE.INC, and recompile TPPICK.PAS.
2. Add an extra video attribute to each PickColorArray (see PickColorArray in the declarations later in this section).
3. In your TPPICK string function, set the TPPICK variable Pickable to False whenever it is processing an item that you want to disable. TPPICK automatically reinitializes Pickable to True for each item, so you only need to set it False for disabled items.

It's worth describing how TPPICK manages cursor movement in the presence of protected items. First, note that it's a programming error to protect every item in the list. If you call PickWindow for such a list, TPPICK won't highlight any items at all and will simply wait until <Esc> is pressed.

If the user presses an arrow key while processing the list and the resulting item is protected, TPPICK enters a special search loop. It repeatedly executes the just-processed command until a) it reaches an unprotected item, b) the item doesn't

change (because it's stuck at the bottom of a column, for example), or c) it reaches the original item again. If no unprotected item is found by this loop, TPPICK exhaustively searches the item list (in a direction consistent with the last command) to find any unprotected item.

You can design degenerate pick lists where this heuristic behavior doesn't seem to make much sense; even so, it gives logical results in almost all cases. One seemingly illogical case occurs when the last item in a list is protected and the user mouse-clicks on the last position of the list's scroll bar. TPPICK attempts to select the last item and fails; repeating the downward action leads it to the first item on the list.

If you attempt to select a protected item with the mouse, TPPICK simply ignores the attempt. Similarly, string searching ignores protected items.

Types

```
PickColorType = (WindowAttr,     {Color for normal unselected items}
                 FrameAttr,      {Color for window frame}
                 HeaderAttr,     {Color for window header}
                 SelectAttr,     {Color for normal selected item}
                 AltNormal,      {Color for alternate unselected items}
                 AltHigh         {Color for alternate selected item}
                 {$IFDEF PickItemDisable}
                 ,
                 UnpickableAttr {Color for unpickable item}
                 {$ENDIF}
                 );
PickColorArray = array[PickColorType] of Byte;
```

PickColorType defines the various colors used within a pick window. The PickColorArray is just a convenient way to collect these colors into a single structure. The colors other than FrameAttr and HeaderAttr may require a little explanation. The pick routines normally display all items using the attribute associated with WindowAttr, with the exception of the currently selected item which is displayed using the SelectAttr color. In addition, the pick routines provide an alternate color set, designated with the elements AltNormal and AltHigh. Any item within the pick list may be designated as "alternate" by setting the boolean typed constant AltPickAttr (described below) to True whenever that item is returned by the user string function. Such items are displayed using the AltNormal attribute when deselected, and the AltHigh attribute when selected. UnpickableAttr is used for items that are protected.

```
PKType = PKSNone..PKSUser9;
```

PKType defines the class of actions performed by the pick routine. It includes such actions as cursor up and down, escape, and item selection. See the constants below for further details.

```
SrchString = string[MaxSearchLen];
```

In combination with the constant MaxSearchLen, this type specifies the longest search string used in an incremental search.

```
SrchType = (NoPickSrch,          {Alphanumeric characters ignored}
            StringPickSrch,      {Incremental search}
            StringAltSrch,       {Alternate incremental search}
            CharPickSrch,        {Single character search}
            CharPickNow);        {Single character search, exit on match}
```

SrchType defines the various searching schemes that TPPICK can employ during the pick. NoPickSrch means that all characters besides the cursor keys, <Enter> and <Esc> are ignored.

StringPickSrch activates an incremental search: when the first alpha character is entered, the highlight bar moves to the next item which starts with that character. Another alpha character moves the highlight bar to the next item that starts with the two characters entered, and so on. This approach is useful for long lists that have been sorted alphabetically. When StringPickSrch is selected, TPPICK will write the current incremental search string in the lower left portion of the window frame, as long as there is sufficient space. StringAltSrch is an alternate form of

incremental search that combines some aspects of the single character searching methods. StringAltSrch works just like StringPickSrch until a character is entered that doesn't match. At this point, the search string is reset and the highlight bar positioned on the next item starting with the new character.

When CharPickSrch is active, entering an alphanumeric character causes the highlight bar to move to the next item that starts with that character. CharPickNow has the same behavior, but it also exits the pick routine, with the next matching item as the final selection.

For all of these search options, character matching is not case-sensitive, and searching is performing in a circular fashion--if no item beyond the current one matches, the search continues at the start of the list.

Constants

```
AltPickAttr : Boolean = False;
```

This option is applied differently than any of the TPPICK constants described so far. Whereas the other constants need only be set once just prior to calling PickWindow, the AltPickAttr constant should be changed within the special string function defined for each pick list. Each time the string function is called to return an item name, it may also set AltPickAttr True. When this is done, TPPICK will display that particular item using the alternate colors defined in the PickColorArray. TPPICK then resets AltPickAttr to False before calling the string function again.

A useful example of the AltPickAttr constant is shown in the TPDIR unit, which displays subdirectory names in different attributes from normal file names.

```
HideCursor : Boolean = True;
```

By default, TPPICK turns off the blinking hardware cursor while a pick window is on-screen, and restores it to the previous state when the pick window is removed. If HideCursor is False, though, TPPICK leaves the cursor shape unchanged while the pick window is displayed.

```
MaxSearchLen = 16;  {Maximum length of incremental search string}
```

This constant specifies the maximum length of any incremental search string. Its value matters only when the PickSrch constant equals StringPickSrch.

```
PickAlterPageRow : Boolean = True;
```

Affects the behavior of <PgUp> and <PgDn> when the pick list contains more entries than will fit in one window. When PickAlterPageRow is False, TPPICK does not change the window-relative row of the highlight bar when <PgUp> or <PgDn> cause the items to scroll under it, until the first or last page is reached. This is useful when the pick list organization assumes fixed page break positions. When PickAlterPageRow is True (the default), pressing <PgDn> always moves the highlight bar to the last row of the window, and <PgUp> moves it to the first row.

```
PickMatrix : Byte = 1;
```

PickMatrix is an important constant that you'll probably want to use often. It specifies how many columns of items will appear within a pick list. For short lists, a

single column is appropriate, but for longer lists such as the files in a directory, many columns may be better. To use the PickMatrix constant, assign it the desired value just prior to calling PickWindow or any of the other TPPICK routines.

Note that TPPICK's display algorithm places items in order going down a column: for an alphabetized list, all of the letter a's might go down the first column, then the b's in the second, and so on.

```
PickMinRows : Word = 0;
PickMaxRows : Word = 9999;
```

PickMinRows and PickMaxRows specify how much leeway TPPICK has in choosing a window size. Although you specify a window height for TPPICK to use when you call PickWindow, TPPICK will adjust the height in case there are fewer items than there were rows specified in the window. Even so, TPPICK always guarantees that the number of rows in the window are no more than PickMaxRows and no less than PickMinRows, no matter what any other parameter or calculation may imply.

```
PickMore : Boolean = True;
```

When the pick list contains more items than will show on one screen, TPPICK can put a prompt in the pick window frame indicating that there are items not currently visible. The "more" prompt does not appear when PickMore is False.

```
PickMousePage : Boolean = False;
```

Affects TPPICK's behavior when the left mouse button is pressed while the mouse cursor is in the upper right or lower right corner of the pick window (i.e., on the scroll bar arrows). The default state of PickMousePage is False. In this case, clicking the mouse makes the pick window act just as if an <Up> or <Down> arrow key was pressed. When PickMousePage is True, clicking makes the pick window act like <PgUp> or <PgDn> was pressed.

```
PickMouseScroll : Boolean = True;
```

When TPPICK's mouse capabilities are enabled (by calling EnablePickMouse), this constant controls whether TPPICK displays and supports a scroll bar. Since the scroll bar appears within the right hand frame of the pick window, it is only available when the window is framed. Beyond that, the scroll bar appears only when there are more items to display than will fit in one window. The scroll bar has several capabilities. At the upper and lower corners of the frame, vertical arrows are displayed. TPPICK scrolls the item display by one row whenever the mouse cursor has been positioned over one of these arrows and the left button pressed. TPPICK also displays a block character within the vertical frame to indicate the approximate position of the current item within the overall list. TPPICK displays a different set of items and adjusts this "slider" when the mouse cursor is placed at a different location within the scroll bar and the left button pressed.

```
PickMouseWindow : Boolean = True;
```

Controls whether the mouse window is restricted to the area of the pick list. When this option is True, as it is by default, the mouse cannot be moved outside of the

pick list while the list is on-screen. If the option is set to False, the mouse can be moved around the entire screen.

```
PickSrch : SrchType = NoPickSrch;
```

This constant controls what form of searching is available within the pick window. See the definition of SrchType above for more details. By default, no alphanumeric searching will occur.

```
PickSrchStat : Boolean = True;
```

Controls display of the incremental search string in the lower window frame. PickSrchStat defaults to True. In this case the current search string appears at the lower left edge of the frame. Otherwise, the search string is not displayed.

```
PickStick : Boolean = True;
```

When the highlight bar reaches the bottom of a column on a short pick list, the bar automatically wraps to the top item. For longer lists (those with more elements than will fit within the window at one time), this behavior can be disconcerting, since it causes the window to scroll at the same time the wrap occurs. When PickStick is True, the wrap behavior is disabled for lists that would scroll. Set it False to allow wrap even in this case.

```
PKSNone = 0;        {Not a command}
PKSAlpha = 1;       {A character used for alphanumeric searching}
PKSUp = 2;          {Cursor up one line}
PKSDown = 3;        {Cursor down one line}
PKSPgUp = 4;        {Cursor up one page}
PKSPgDn = 5;        {Cursor down one page}
PKSLeft = 6;        {Cursor left one column}
PKSRight = 7;       {Cursor right one column}
PKSExit = 8;        {Exit the pick list without making a selection}
PKSSelect = 9;      {Select current item and exit pick list}
PKSHelp = 10;       {Invoke help routine}
PKSHome = 11;       {Move cursor to first item}
PKSEnd = 12;        {Move cursor to last item}
PKSProbe = 13;      {Check the mouse position and perform an action}
PKSUser0 = 14;      {User-defined exit commands}
...
PKSUser9 = 23;
```

These are the codes for each of TPPICK's cursor movement and selection commands. Most of them are self-explanatory. The last ten (PKSUser0.. PKSUser9) are set aside to allow user-defined "exit commands" to be implemented. If one of these commands is executed, TPPICK will do the same things that it would if <Enter> were pressed; it will then be up to you to check the global variable PickCmdNum and perform the function associated with the command just issued. For more details on how to implement user-defined commands, see Appendix B.

```
SrchStart : Byte = 1;
```

When any form of alphanumeric searching is enabled, this option controls where the search starts within each item string. By default, searching starts with the first character of each item.

Variables

```
PickChar : Char;
```

Last key entered in TPPICK. For extended keystrokes, PickChar holds the scan code of the keystroke OR $80, that is, with the high bit set. Note that PickChar will hold the actual keystroke entered when the CharPickNow option is selected and an alphanumeric keystroke has selected a particular item.

```
PickCmdNum : PKType;
```

The last pick command given to TPPICK. Can be used to determine whether the user selected an item (PickCmdNum = PKSSelect) or escaped without making a selection (PickCmdNum = PKSExit). Note that PickCmdNum will equal PKSSelect when the CharPickNow option is selected and an alphanumeric keystroke has been entered to select a particular item. Checking PickCmdNum is preferable to PickChar, in case the keystrokes for each command have been reassigned.

```
PickHelpPtr : Pointer;
```

Points to a routine to be called when the help command is given. See Appendix B for details.

```
PickKeyPtr : Pointer;
```

Points to a routine that will return the next keystroke. See Appendix B for details.

```
PickUserPtr : Pointer;
```

Points to a routine that will be executed just after the screen has been updated each time a pick keyboard command has been entered. Note that the use of this routine is optional; it is not the same as the required string function whose address is passed as a parameter to PickWindow. If not nil, this variable must point to a procedure compiled under the FAR model, and with the following form, where Item is the index of the currently selected item:

```
procedure UserPick(Item : Word);
```

PickUserPtr may be useful when the movement of the pick highlight bar should cause some other screen update to occur outside of the pick window itself.

DisablePickMouse

Declaration

```
procedure DisablePickMouse;
```

Purpose

Disable mouse control of pick lists.

Comments

This routine undoes the actions of EnablePickMouse. For many programs, you'll never need to call DisablePickMouse, since its essential functions are handled automatically by a Turbo Pascal exit routine. In some cases, particularly in memory resident programs, you may need to call DisablePickMouse in order to disable the mouse functions temporarily.

Note that DisablePickMouse does not hide the mouse cursor. You can call HideMouse to do that if appropriate.

Examples

```
{TSR pops up}
...
EnablePickMouse;
if PickWindow(...) then begin
  ...
end;
DisablePickMouse;
...
{TSR returns}
```

See Also

EnablePickMouse

Declaration

```
procedure EnablePickMouse;
```

Purpose

Enable mouse control of pick lists.

Comments

Call this routine to activate mouse control of pick lists. The routine checks to assure that a mouse is installed, then sets an internal flag so that other pick routines will utilize the mouse. You should call EnablePickMouse once prior to calling TPPICK's PickWindow routine.

Since TPPICK's mouse control depends on the TPMOUSE mouse event handler, EnablePickMouse also calls TPMOUSE's EnableEventHandling routine. There is no harm done if you've called this routine previously. Be aware, though, that any other mouse event handler which you've previously activated will lose control after the call to EnablePickMouse.

EnablePickMouse does not modify the current mouse window, the mouse cursor type, or the mickey-to-pixel ratio, nor does it display the mouse cursor. Your program should set those values as needed by calling the appropriate TPMOUSE routines. You may use either a hardware or software mouse cursor.

When the mouse is enabled, TPPICK performs the following actions during pick list selections. When the left mouse button is released, TPPICK moves the highlight bar to the item over which the mouse cursor is currently positioned. Releasing the left button a second time with the mouse cursor over the highlighted item causes the pick routine to exit with that selection. This "double click" requirement provides better user feedback and control than immediate selection by a single click. Pressing the right button while in a pick window is equivalent to pressing the <Esc> key, exiting the window without making a selection. Pressing both buttons is equivalent to pressing <F1>, activating a user-defined help routine if specified.

If the pick window has a frame and the TPPICK typed constant PickMouseScroll is True, the mouse can also be used to scroll within pick lists that are longer than the window. See the description of PickMouseScroll for details.

The actions performed when mouse buttons are activated may be configured in the same way that keystrokes are installed. See Appendix B for details.

Note that mouse actions occur when the buttons are *released*, not when they are first pressed. This technique allows reliable use of commands that require both buttons to be pressed simultaneously. Change this behavior by toggling the WaitForButtonRelease constant in TPMOUSE. After that change, mouse actions occur immediately after a mouse button is depressed.

In a memory resident program using TPPICK, you should not enable the mouse prior to going resident. Instead, you should enable the mouse after popping up, and disable it again before returning. There are additional considerations when writing a memory resident program that uses the mouse. See Chapter 2, Section G for details.

EnablePickMouse

Examples

```
begin
  ClrScr;
  WriteLn('Select a pizza topping');

  TpPick.PickSrch := CharPickNow;        {Accept first letter name selections}

  TpPick.PickMouseScroll := True;        {Allow mouse scrolling}
  EnablePickMouse;                       {Enable the mouse in the pick window}

  Choice := 1;                           {Set initial choice}

  if not PickWindow(@PizzaTopping,       {Routine that returns item names}
                    NumPizzaToppings,    {Number of items in pick list}
                    5, 5, 17, 11,        {Coordinates of the window}
                    True,                {Draw a frame around the list}
                    PickColors,          {Colors to use}
                    '',                  {No title for this window}
                    Choice)              {Which item was chosen}
  then
    {Insufficient memory for screen buffer}
    WriteLn('Unable to execute PickWindow')
  else if PickCmdNum = PKSSelect then
    {A selection was made}
    WriteLn('You chose:', PizzaTopping(Choice))
  else
    {Escape was pressed}
    WriteLn('No choice was made');
end.
```

Shows the two-line modification to the introductory example required to enable full mouse functions.

See Also

DisablePickMouse

EnableEventHandling (TPMOUSE)

Declaration

```
procedure FillPickWindow(W : WindowPtr;
                         StringFunc : Pointer;
                         NumItems : Word;
                         Colors : PickColorArray;
                         Choice : Word;
                         FirstChoice : Word);
```

Purpose

Display a predefined window, fill it with choices, and return.

Comments

FillPickWindow performs a portion of the function of the central routine PickWindow. FillPickWindow is generally used in combination with PickBar, described later. This combination is preferable to PickWindow when more than one selection must be made from a single pick list. Whereas PickWindow draws and erases the pick window each time it is called, PickBar uses a window previously initialized by FillPickWindow, and leaves that window on-screen when it exits. This avoids an unsightly screen flash that will occur if PickWindow is called repeatedly in the same location.

The window W must be initialized by a call to MakeWindow (TPWINDOW) prior to calling FillPickWindow. The window W may already be on screen at the time FillPickWindow is called, but if so, it must be the current window. If W is not currently active, FillPickWindow will display it.

StringFunc is the address of a function that will return the name of each pick item. See the introduction to this section, and the entry for PickWindow, for more details. NumItems specifies the number of items on the pick list, indexed from 1 to NumItems. Colors specifies the colors to use while drawing the pick list.

Choice specifies which item to highlight initially, and Row specifies the item number that appears in the upper left hand corner of the window.

Note that FillPickWindow updates the screen and returns immediately without waiting for keystrokes. It leaves the specified window active, thus making any cursor coordinates relative to the pick window boundaries.

Be sure to set the values of any of TPPICK's display constants (PickMatrix, etc.) prior to calling FillPickWindow.

Examples

```
var
  W : WindowPtr;
  Choice : Word;
  Row : Word;
begin
  {Define the window first}
  if not MakeWindow(W, 5, 5, 17, 11, True, False, False,
                    PickColors[WindowAttr], PickColors[FrameAttr],
                    PickColors[HeaderAttr], '') then Exit;
```

```
  {Fill it with items}
  Choice := 1;
  Row := 1;
  FillPickWindow(W, @PizzaTopping, NumPizzaToppings,
                 PickColors, Choice, Row);

  {Get choices until <Esc> is pressed}
  repeat
    PickBar(W, @PizzaTopping, NumPizzaToppings,
            PickColors, False, Choice, Row);
    if PickCmdNum = PKSSelect then
      {A selection was made}
      FastWrite('Last choice was:'+Pad(PizzaTopping(Choice), 20), 15, 1, $07);
  until PickCmdNum <> PKSSelect;

  {Erase and dispose of the window}
  DisposeWindow(EraseTopWindow);
end.
```

A variation on the example at the beginning of this section, showing how PickBar can be used to get many choices from one window. First, the window is defined with a call to MakeWindow. Next, FillPickWindow draws the window with an initial set of choices. PickBar is called repeatedly, and the current selection is written to the screen. When the user presses <Esc>, the loop ends and the window is erased from the screen and deallocated.

See Also

MakeWindow (TPWINDOW) PickBar

Declaration

```
procedure PickBar(W : WindowPtr;
                  StringFunc : Pointer;
                  NumItems : Word;
                  Colors : PickColorArray;
                  EraseBar : Boolean;
                  var Choice : Word;
                  var FirstChoice : Word);
```

Purpose

Choose from a pick list already displayed on the screen.

Comments

PickBar works in concert with FillPickWindow to allow multiple selections from a single window. See FillPickWindow for details.

The parameters to PickBar are the same as those to FillPickWindow, with the following additions. When EraseBar is true, PickBar recolors the selected item as if it were unselected before exiting the routine.

Choice is used first to set the initially highlighted selection. When PickBar exits, Choice holds the index of the selected item. Row is the item number that appears in the upper left hand corner of the window.

Be sure to set the values of any TPPICK search constants (PickSrch, etc.) prior to calling PickBar.

PickBar and PickWindow are reentrant. If the selection of one item leads to the display of another pick list, you may safely change the values of any of TPPICK's global constants before displaying the second pick list. TPPICK automatically saves and restores its state upon entry and exit. You must erase and dispose of each window allocated for FillPickWindow before returning to a previous pick list.

Examples

See the example for FillPickWindow.

See Also

FillPickWindow

Declaration

```
{$IFDEF EnablePickOrientations}
function PickOrientation : PickOrientType;
```

Purpose

Return the current pick list orientation.

Comments

This routine determines which pick orientation is currently active. PickOrientType is an enumerated type containing the values PickOrientNone (returned if no orientation is selected), PickVertical, PickHorizontal, and PickSnaking.

PickWindow

Declaration

```
function PickWindow(StringFunc : Pointer;
                    NumItems : Word;
                    XLow, YLow : Byte;
                    XHigh, YHigh : Byte;
                    DrawFrame : Boolean;
                    Colors : PickColorArray;
                    Header : string;
                    var Choice : Word
                    ) : Boolean;
```

Purpose

Display a pick window, let user scroll around in it, and return choice.

Comments

PickWindow combines the functions of FillPickWindow and PickBar into one easy operation. PickWindow defines a window, displays it, fills the window with pick items, lets the user browse through the list, and returns a selection.

StringFunc is the address of a function which returns the name of each pick item. This function must be compiled under the FAR model, and must be declared with the following type:

```
{$F+}                    {Must be FAR}
function ItemName(Item : Word) : string;
  {-Return the name of each item}
begin
  ItemName := ...        {Must return an item name}
  AltPickAttr := ...     {Optional}
end;
{$F-}
```

Each time PickWindow displays an item, it calls the specified string function to get the name of the item. PickWindow passes the routine an index in the range 1..NumItems to specify the item. The string returned by the string function is displayed unchanged, with one exception: blanks are appended to reach the required width. Besides returning an item name, the string function may set one additional variable which will affect the appearance of the display. If the TPPICK constant AltPickAttr is set to True, the item will be displayed using the alternate colors (as described in PickColorType, above).

Because the string function is called quite often during list scrolling, we recommend that it perform the least amount of computation possible on each call. If possible, precompute the strings to be returned. See TPDIR for an example: the file directory is read once only, and the strings holding formatted names, dates, times and sizes are generated once before calling PickWindow.

XLow, YLow, XHigh, and YHigh determine the window coordinates where the pick list will appear. These values include space for the window frame, depending on whether DrawFrame is True. The width available for the display of each pick item is determined (approximately) by (XHigh-XLow) div PickMatrix, where PickMatrix is the number of columns of items. It is your responsibility to specify values of XLow, XHigh, and PickMatrix that allow enough width for each item to

appear. TPPICK will truncate or pad each item name to fit in the allotted space.

PickWindow may also adjust the value of YHigh that you specify. If the number of pick items is smaller than the number of rows available between YLow and YHigh, PickWindow will reduce YHigh. If the number of rows available exceeds PickMaxRows or is less than PickMinRows, it will adjust YHigh. If the number of pick items exceeds the number of screen rows, TPPICK activates vertical scrolling.

A Header, if specified, will appear centered at the top of the window frame.

Choice specifies which item should be highlighted initially, and upon return from PickWindow holds the index of the selected item.

PickWindow returns True if the window was successfully displayed, and if the parameters passed to the function were acceptable. If it returns False, this should be considered a program error, either because insufficient heap space was available, or because faulty parameters were passed. PickWindow requires heap space large enough to buffer the screen area covered by the window, plus a small added amount for a window stack.

Upon successful return from PickWindow, the TPPICK global variables PickCmdNum and PickChar hold the command used to exit the pick function and the final keystroke, respectively. We recommend that applications check the value of PickCmdNum rather than PickChar, since the keystrokes can be reinstalled with different values. The following values are returned in PickCmdNum when PickWindow exits:

```
PKSExit          {Exit the pick list without making a selection}
PKSSelect        {Select current item and exit pick list}
PKSUser0         {User-defined exit commands}
...
PKSUser9
```

The TPHELP unit makes use of the user-defined exit command PKSUser9 to activate the "previous help topic" function while picking from help index. You should avoid redefining PKSUser9 if your application also uses TPHELP.

Be sure to set the values of any of TPPICK's constants (PickMatrix, PickSrch, etc.) prior to calling PickWindow.

Note that TPWINDOW options (Shadow, Explode, etc.) active at the time PickWindow is called will affect the appearance of the pick window.

Examples

See the beginning of this section for a simple yet complete example of using PickWindow. See the TPDIR and TPHELP units as well as the ENTRY demo program for more complex examples.

See Also

FillPickWindow
MakeWindow (TPWINDOW)
PickBar

SetHorizontalPick / SetSnakingPick

Declaration

```
{$IFDEF EnablePickOrientations}
procedure SetHorizontalPick;
```

Purpose

Select a horizontal pick list orientation.

Comments

If the conditional define EnablePickOrientation is activated in TPDEFINE.INC, this routine is included in TPPICK. SetHorizontalPick changes the pick list orientation to horizontal. See "Pick List Orientations" earlier in this section for a description of the possible orientations.

Declaration

```
{$IFDEF EnablePickOrientations}
procedure SetSnakingPick;
```

Purpose

Select a snaking pick list orientation.

Comments

If the conditional define EnablePickOrientation is activated in TPDEFINE.INC, this routine is included in TPPICK. SetSnakingPick changes the pick list orientation to snaking. See "Pick List Orientations" earlier in this section for a description of the possible orientations.

Declaration

```
{$IFDEF EnablePickOrientations}
procedure SetVerticalPick;
```

Purpose

Select a vertical pick list orientation.

Comments

If the conditional define EnablePickOrientation is activated in TPDEFINE.INC, this routine is included in TPPICK. SetVerticalPick changes the pick list orientation to vertical. See "Pick List Orientations" earlier in this section for a description of the possible orientations.

I. TPDIR: Popup Directory Picker

TPDIR builds upon the pick unit TPPICK to solve a common programming problem: the selection of filenames. The single interfaced routine of TPDIR, GetFileName, takes a wildcard specification as input, opens a window displaying the matching files, allows the user to browse around, and finally returns the selected filename.

By taking advantage of the flexibility inherent in the TPWINDOW, TPPICK and TPDATE units, TPDIR provides all the options you could want. The directory window may be of any shape, size, and color, and will explode or implode on command. Files may be displayed in one or more columns, with or without date, time, and size in a format that you can specify. Within the pick list, files may be selected by first letter or using an incremental search. TPDIR even lets the user browse through multiple subdirectories by pointing at the directory name and pressing a key.

Using TPDIR is easy, as demonstrated by the following example:

```
{$V-}
program DirExample;
uses
  Dos, TpString, TpCrt, TpWindow, TpCmd, TpPick, TpDir;
const
  DirColors : PickColorArray = ($07, $07, $0F, $70, $0F, $78);
var
  Mask : string[79];
  FName : string[79];
begin
  ClrScr;

  TpPick.PickSrch := StringPickSrch; {Activate incremental searching}
  TpDir.ShowSizeDateTime := True;    {Show size, date and time of files}

  repeat
    Write('Enter a filename mask: ');
    ReadLn(Mask);
    case GetFileName(Mask,          {The filename mask}
                     AnyFile,       {Match any file type}
                     1, 3,          {Position the directory window}
                     15, 1,         {Use one column, ending at row 15}
                     DirColors,     {Specifies colors to use}
                     FName          {Which filename was chosen}
                     ) of
      0 : if Length(FName) = 0 then
            WriteLn('You pressed <Esc>')
          else
            WriteLn('You chose=', FName);
      1 : WriteLn('Path not found');
      2 : WriteLn('No matching files');
      3 : WriteLn(FName, ' is a new file');
```

```
        4 : WriteLn('Insufficient memory');
        5 : WriteLn('Window does not fit on screen');
        6 : WriteLn('No orientation selected');
      else
        WriteLn('Dos critical error');
      end;
    until Length(Fname) = 0;
  end.
```

DirColors takes the same type as the normal TPPICK color array. Although we've specified a single value here, you'll usually want to have two color arrays, one of which you select on a color monitor and the other on monochrome.

Mask is the DOS wildcard string for the files that GetFileName should match. More details on this below. FName is the selected filename. GetFileName always returns a complete pathname, including drive, directory, name, and extension.

To start the program, we set two constants that customize the behavior of TPPICK and TPDIR. In the example, we activate incremental searching in TPPICK and enable the display of file size, date and time in TPDIR.

Next, we enter a loop to read the file mask, then use GetFileName to pop up a directory window. GetFileName returns an error code so that we can decide what happened in the routine.

That's it! For more details about TPDIR's flexible options, read on.

TPDIR: Declarations

Constants

```
DatePicture : string[12] = 'Mm/dd/yy';
```

This string is a picture mask (as defined in TPENTRY and TPDATE) for the date display. If the string is empty, the date will not appear. Otherwise, the mask characters defined in TPDATE may be combined to format the date as desired. With the default DatePicture, dates will appear as follows:

```
 2/02/88
12/29/87
```

```
DirsUpper : Boolean = True;
```

If True, subdirectory names are displayed in uppercase characters, otherwise in lowercase.

```
DirDisplayStr : string[5] = '<dir>';
```

This string is displayed in place of the file size for subdirectory entries.

```
FilesUpper : Boolean = False;
```

If True, normal file names are displayed in uppercase characters, otherwise in lowercase.

```
SeparateDirs : Boolean = True;
```

If True, subdirectory names are sorted to the beginning of the directory list. Otherwise, they are displayed in alphabetic order, mixed with the normal files.

```
ShowExtension : Boolean = True;
```

If True, the three-character file extensions are displayed as part of each file name. Setting this constant to False is appropriate only when the directory window will hold a group of files all having the same extension.

```
ShowSizeDateTime : Boolean = False;
```

If True, the size, date and time of files may be displayed in the directory window, depending on additional constants described just below. If False, only the file name is shown.

```
SizeDisplay : (SizeNone,        {Don't display the file size}
               SizeBytes,       {Display size in bytes}
               SizeKBytes)      {Display size in kilobytes}
               = SizeBytes;
```

The size may be omitted, displayed in bytes, or adjusted to the nearest kilobyte.

```
TimePicture : string[12] = 'Hh:mmt';
```

This string is a picture mask for the time display. If the string is empty, no time will appear. Otherwise, the mask characters defined in TPDATE may be combined to format the time as desired. With the default TimePicture, times will appear as follows:

```
 6:12a
11:55p
```

```
UseFileFrame : Boolean = True;
```

If True, a frame is drawn around the directory window, using the current frame stored in TPCRT.FrameChars.

```
ReturnCompletePath : Boolean = True;
```

If True, complete pathnames for files are returned.

ChangeDirectory

Declaration

```
function ChangeDirectory(Mask : string;
                         XLow, YLow, YHigh, PickCols : Byte;
                         Colors : PickColorArray) : Word;
```

Purpose

Change to a new drive or directory.

Comments

This routine provides a convenient method for the user of a program to switch to a different drive or directory.

If Mask is an empty string or contains wild cards, ChangeDirectory will pop up a list of all the subdirectories within the specified path. Whenever the user presses <Enter> to select one of those directories, ChangeDirectory changes the current directory to match the selection, and displays a list of all the subdirectories there. When the user finally presses <Esc>, the current directory will be set to the last selected directory.

If Mask does not contain wildcards, ChangeDirectory simply changes to the specified directory and returns without displaying a window.

The remaining parameters to ChangeDirectory have the same meaning as those passed to TPDIR's GetFileName routine.

ChangeDirectory returns a status word. Its value is zero for success, and will typically hold a Turbo Pascal runtime error code for failure. There are two exceptions. ChangeDirectory returns the following special error codes for consistency with GetFileName:

4 Insufficient memory
5 Won't fit on screen
6 No pick orientation

The most common runtime error codes will be:

3 Path not found (non-existent subdirectory)
15 Invalid drive

Declaration

```
function GetFileName(Mask : string;
                     FileAttr : Byte;
                     XLow, YLow : Byte;
                     YHigh, PickCols : Byte;
                     Colors : PickColorArray;
                     var FileName : string
                     ) : Word;
```

Purpose

Given a Mask (which may or may not contain wildcards), pop up a directory window, let user choose, and return pathname. Return a status code.

Comments

If you pass an empty string, GetFileName equates that to '*.*' and pops up a directory window. If you pass a complete filename, GetFileName checks to see whether the file exists, and returns a status code without popping up a window. If you pass the name of a drive or subdirectory, GetFileName displays a window with all the files in that location. The rules for specifying Mask match those for DOS wildcards.

FileAttr determines which files the directory search will find. Acceptable values for FileAttr are defined within the Turbo Pascal DOS unit. If the Directory bit ($10) of FileAttr is set, GetFileName will display subdirectories in its list, and allow the user to select subdirectories to browse around a hard disk. If the Directory bit is not set, only normal files will be displayed, and the user will be restricted to browsing a single directory. By selecting an appropriate value of FileAttr, you can decide whether read-only, hidden, and system files will appear in the directory display.

XLow and YLow specify the upper left corner of the directory window, including the frame if activated. PickCols determines the number of columns of filenames in the pick window. If size, date and time are displayed, you'll struggle to fit two columns of filenames on-screen. Otherwise, you can have up to five columns of names. If you specify a value of PickCols that doesn't fit, GetFileName will adjust it for you. YHigh specifies the highest row for the directory window. This value is subject to adjustment by TPPICK. See section 2.H for details.

Colors determines the video attributes for the directory. Normal files are displayed in the attributes associated with WindowAttr and SelectAttr (when selected). Subdirectories are displayed using AltNormal and AltHigh (when selected).

GetFileName determines the window width to use based on the number of columns of filenames, the filename format, and the search mask displayed at the top of the window. It picks the wider of the search mask and the filenames in an attempt to show everything. There is a catch, however: the search mask it uses to choose the width is identical to the one you pass to GetFileName. Later, GetFileName expands the search mask to include the complete pathname (drive and directory as well as wildcards) and it displays this complete pathname as the window title. If, for example, you've passed '*.*' for the search mask, you're in the directory 'c:\pascal\test', and you're using only one column of filenames, the search mask will be truncated when it is displayed. To avoid this problem, expand the search mask to a complete pathname before calling GetFileName.

GetFileName

If ReturnCompletePath is True, GetFileName returns a fully instantiated path name in all cases, even if the specified file is in the current directory. When ReturnCompletePath is set False, GetFileName returns a relative path name. However, as soon as the user navigates to other subdirectories with the TPDIR menus, GetFileName reverts to returning a complete path name. This behavior is required to keep the relative path name from growing longer than 79 characters after an extended navigation session.

If an error occurs in GetFileName, or if the user presses <Esc>, FileName will be an empty string.

GetFileName returns a status code in the function result. Here are the values:

0 Success
1 Path not found
2 No matching files
3 New file
4 Insufficient memory
5 Won't fit on screen
6 No pick orientation
else a Turbo Pascal critical error code

A value of 1 indicates that an invalid drive or subdirectory name was specified in the input mask. Code 2 means that no files were found that matched the specified mask and attribute. Code 3 means a valid and complete file name was given in the input mask, but no file with that name currently exists. Code 4 means that there was not enough heap space for GetFileName's internal use (more on this below). Code 5 means that the specified window coordinates were incorrect--not even one column of filenames could be displayed with the given coordinates. Code 6 means that TPPICK was not configured for a pick orientation. Other non-zero values of the status code mean a Turbo Pascal critical error. These values will be returned after an attempt to get a file on an floppy drive with no disk loaded, for example.

TPDIR needs heap space for two reasons. First, TPDIR allocates heap space for window buffers. The amount depends primarily on the size of the window. Second, it temporarily allocates space to store an array of filenames. The size per file depends on whether the size, date, and time are displayed--it ranges from a minimum of 16 bytes per file with no date, time or size to a maximum of about 50 bytes with all picture masks specified. TPDIR will allocate space for up to 500 files in any directory. If less space is available, it will allocate that much and ignore any files in excess of the number allocated. Only if TPDIR can allocate space for fewer than 2 filenames will it exit with a status code of 4.

GetFileName always displays the filenames sorted alphabetically. Depending on the state of SeparateDirs, the subdirectory names will be mixed in with or separated from the regular files.

GetFileName stores the state of TPPICK's PickMatrix global constant before modifying it, and restores it prior to exiting.

Examples

```
Status := GetFileName('*.DAT', 0, XLow, YLow, YHigh, 1, Colors, FileName);
```

Displays all normal files in the current directory with extension .DAT.

Subdirectories are not displayed, so the user cannot look in other directories.

```
Status := GetFileName('*.DAT', Directory, XLow, YLow, YHigh, 1, Colors,
                      FileName);
```

Displays all normal files in the current directory with extension .DAT. Subdirectories *are* displayed, so the user can browse around to other directories if desired.

See the example at the beginning of this section for a complete working program that uses TPDIR. See the PTRPT demo program for the use of TPDIR in a more complex setting.

See Also

Date and Time pictures (TPDATE) PickWindow (TPPICK)

J. TPSCREEN: Virtual Screens

The main attraction in the TPSCREEN unit is its support for *virtual screens*. Working in conjunction with the TPCRT unit, the routines here allow you to treat almost any region of memory as a "screen", write to it, and then move its contents to the actual screen. Normally a virtual screen is either an alternate video page (on a color graphics adapter) or a buffer allocated on the heap.

TPSCREEN lets you define as many as 10 (MaxScreens) virtual screens at any one time, not including the actual screen, and the number you assign to a screen is referred to as its *handle*. (The actual screen is always screen #0.) You can define a screen in one of two ways. The usual method is to call AllocateScreen with a handle and the number of rows and columns in the screen, and allow it to manage the allocation and deallocation of buffer space for you. If you want to do your own memory management, you can call DefineScreen instead, passing it a pointer to the buffer. This would allow you, for example, to use alternate video pages for buffers when running on a machine with a color adapter, and allocate heap space only when running on a monochrome system. Keep in mind that any screen I/O done during Read's and ReadLn's is always directed to the visible page (see below) of the actual screen, so the user can see what's being entered.

If you know that your program will be running only on color systems, you have another option as well. TPSCREEN draws a distinction between the *active display page* and the *visible display page*. The visible page is the one currently seen on the monitor. The active page is the one where screen I/O is currently being directed. Consider the following code, for example:

```
SetActivePage(1);
ClrScr;
SpeedWrite('Hello world!');
SetVisiblePage(ActivePage); {= 1}
```

The first line tells TPSCREEN to send all screen output to video page 1. The second and third lines clear that page and write a string to it. And the fourth line makes the active page the visible page.

Another noteworthy feature of TPSCREEN is *color mapping*. One of the most common problems in screen design is making programs that look good on color monitors look equally good on monochrome and composite (black and white) monitors. The special versions of TextColor and TextBackground in TPSCREEN attempt to get around this problem wherever possible by translating all color combinations into some combination of black and white when your program is run on a system without a color monitor. (See the entry for the MapColor routine in TPCRT as well.)

For examples of how to use the principal routines in TPSCREEN, you should examine the sample program WINEXMPL.PAS, which demonstrates both the speed of TPSCREEN's virtual screens and the benefits of color mapping.

Some Important Restrictions

When you use TPSCREEN in a program, there are a couple of very important points to keep in mind, especially if you are integrating it into an existing program:

First, TPSCREEN performs no magic. It cannot force all screen output to be sent to a virtual screen or to the currently active video page. The screen writing routines in TPCRT--FastWrite, FlexWrite, etc.--will always write to the physical screen, and to the current *visible* display page. If you want to write to virtual screens and non-visible display pages, you must use either Write/WriteLn or one of the routines in TPSCREEN.

The second point is really just a corollary to the first: When you use TPSCREEN in a program, you should place a 'USES TPSCREEN' in all units that perform screen I/O, and TPSCREEN should always appear *after* TPCRT in the USES list. The reason for this is that TPSCREEN must duplicate many of the basic functions of TPCRT--Window, ClrScr, GotoXY, etc.--and it is imperative that these routines be called rather than the ones of the same name in the TPCRT unit.

When NOT to Use TPSCREEN

A number of users have in the past experienced problems caused by using TPSCREEN when they don't need to. A couple of rules of thumb:

- Don't use TPSCREEN if you don't need virtual screens
- Don't try to use TPSCREEN if you are using TPWINDOW or TPMENU

Like TPWINDOW and TPMENU, TPSCREEN "sits on top of" TPCRT, adding advanced functions and capabilities. Unlike the other two units, however, it also takes control of basic screen I/O away from TPCRT. When used, TPSCREEN handles all Write and WriteLn statements, as well as basic functions such as cursor movement and, most notably, windowing. TPWINDOW and TPMENU (which depends on TPWINDOW) expect and demand TPCRT to be in control of these functions, and for this reason they cannot be used with TPSCREEN. To enforce this rule, TPSCREEN now checks for the presence of TPWINDOW, and vice versa. If TPWINDOW is present in a program that USES TPSCREEN, the program will be aborted before it starts.

On a related note, several early users of Turbo Professional 4.0 were using TPSCREEN only because they needed two routines, StoreWindowCoordinates and

RestoreWindowCoordinates. For this reason, we have added these routines (as well as the type declaration for WindowCoordinates) to TPCRT. These routines are identical in function to those in TPSCREEN. We have also added several other routines to both TPCRT and TPWINDOW in order to reduce the temptation to use TPSCREEN when it isn't necessary.

Routines Also Found in Turbo's CRT Unit

In general, the following routines are functionally identical to the routines of the same name in Turbo's CRT unit. The only special points worth noting are:

1. ClrScr and ClrEol can both use a background character other than ' '. See the documentation for SetBackgroundChar.
2. If the global typed constant Turbo3StyleColors is True, HighVideo, NormVideo, and LowVideo behave as they did in Turbo 3.x.
3. As indicated earlier, TextColor and TextBackground attempt to remap colors for improved appearance on monochrome displays.
4. When the active screen is a virtual screen, Window will not allow you to create a window larger than the dimensions of the virtual screen.

```
procedure ClrScr;
  {-Clear the active window. Uses special background character, if not ' '.}

procedure ClrEol;
  {-Clear to end of line. Uses special background character, if not ' '.}

procedure GoToXY(X, Y : Byte);
  {-Move cursor to column X, row Y, relative to Window}

procedure HighVideo;
  {-Select high intensity}

procedure LowVideo;
  {-Select low intensity}

procedure NormVideo;
  {-Select 'normal' intensity}

procedure TextBackGround(Color : Byte);
  {-Set background color for screen writes}

procedure TextColor(Color : Byte);
  {-Set foreground color for screen writes}

procedure TextMode(Mode : Word);
  {-Switch to/set text mode}

function WhereX : Byte;
  {-Return column coordinate of cursor, relative to Window}

function WhereY : Byte;
  {-Return row coordinate of cursor, relative to Window}

procedure Window(X1, Y1, X2, Y2 : Byte);
  {-Set current window coordinates}
```

Routines Also Found in the TPCRT Unit

In general, the following routines are functionally identical to the routines of the same name in TPCRT. The only special points worth noting are:

1. GotoxyAbs does nothing if the active screen is a virtual screen and X,Y is outside the active window.
2. ScrollWindowUp and ScrollWindowDown always affect the *active page* of the physical screen, even if the current screen is a virtual screen.

```
procedure FrameWindow(LeftCol, TopRow, RightCol, BotRow, FAttr, HAttr : Byte;
                      Header : string);
  {-Draws a frame around a window}

procedure GotoXYAbs(X, Y : Byte);
  {-Move cursor to absolute column X, row Y. Does nothing if the active screen is
    a virtual screen and X,Y is outside the active window.}

procedure ScrollWindowDown(X1, Y1, X2, Y2, Lines : Byte);
  {-Scrolls the designated window down the specified number of lines. Always
    affects the active page of the physical screen.}

procedure ScrollWindowUp(X1, Y1, X2, Y2, Lines : Byte);
  {-Scrolls the designated window up the specified number of lines. Always
    affects the active page of the physical screen.}

function WhereXY : Word;
  {-Return absolute row and column coordinates of cursor. High byte has current
    row (Y), low byte has current column (X).}

function WhereXAbs : Byte;
  {-Return absolute column coordinate of cursor}

function WhereYAbs : Byte;
  {-Return absolute row coordinate of cursor}
```

TPSCREEN: Declarations

Constants

```
Black : Byte = 0;
Blue : Byte = 1;
Green : Byte = 2;
Cyan : Byte = 3;
Red : Byte = 4;
Magenta : Byte = 5;
Brown : Byte = 6;
LightGray : Byte = 7;
DarkGray : Byte = 8;
LightBlue : Byte = 9;
LightGreen : Byte = 10;
LightCyan : Byte = 11;
LightRed : Byte = 12;
LightMagenta : Byte = 13;
Yellow : Byte = 14;
White : Byte = 15;
```

By redefining these identifiers (already declared in TPCRT) as typed constants, TPSCREEN is able to remap colors for monochrome and composite black-and-white displays.

```
MaxScreens = 10;
```

Maximum number of virtual screens that TPSCREEN can manage.

```
Turbo3StyleColors : Boolean = False;
```

If True, HighVideo/NormVideo/LowVideo behave as they did under Turbo version 3.x (NormVideo selects yellow on black, etc.).

Types

```
WindowCoordinates = record
                      XL, YL, XH, YH : Byte;
                    end;
```

Record used to save/restore window coordinates.

```
ScreenRec =
record
  Alloc : Boolean; {is it allocated ?}
  Size : Word;     {Size in words}
  SCols,           {Number of columns}
  SRows,           {Number of rows}
  {Window coordinates}
  XLo,             {0..79 format}
  YLo,             {0..24/42/49 format}
  XHi,             {1..80 format}
  YHi,             {1..25/43/50 format}
  Col,             {Cursor coordinates}
  Row : Byte;
  BPtr : Pointer;  {Pointer to screen buffer}
end;
```

This record describes a given screen, either virtual or physical. Alloc indicates whether or not a buffer has been allocated for it dynamically. Size contains the total size of the screen buffer in words. SCols and SRows contain the height and

width of the screen. XLo, YLo, XHi, and YHi contain the XY coordinates for the screen's current window. Col and Row contain the current position (absolute) of the cursor. BPtr points to the screen's buffer.

Variables

```
ActivePage : Byte;
```

The video display page currently being written to.

```
CurrentScreen : Byte;
```

Handle number for the active screen (0 = physical screen).

```
OutputToScreen : Boolean;
```

True if output is going to the physical screen. For internal use only.

```
Screens : array[0..MaxScreens] of ScreenRec;
```

Array of virtual screen descriptors.

```
VisiblePage : Byte absolute CurrentPage;
```

The display page actually visible on screen (same as CurrentPage, in TPCRT).

AllocateScreen / AssignCrt

Declaration

```
function AllocateScreen(Handle, Cols, Rows : Byte) : Boolean;
```

Purpose

Defines a virtual screen and allocates a buffer for it.

Comments

This routine defines a virtual screen (Handle) of the specified dimensions and allocates a buffer for it. Except in cases where you are managing your own screen buffers (using DefineScreen), AllocateScreen needs to be called before any other operations are performed on a virtual screen.

If a buffer was previously allocated for the same screen, it is released before the new one is allocated. Handle must be in the range 1..MaxScreens.

A virtual screen created with TPSCREEN cannot be larger than 64KB. This means that Rows*Cols*2 must be less than 65521. If it isn't, AllocateScreen returns False.

Examples

```
Success := AllocateScreen(0, CurrentCols, CurrentRows);
```

Success will be set to False. Screen 0 (the physical screen) never needs to have a buffer allocated for it.

```
Success := AllocateScreen(1, CurrentCols, CurrentRows);
```

Defines Screen 1 as having the same dimensions as the active screen. Success will be set to False only if a screen buffer of the desired size cannot be allocated.

See Also

DeallocateScreen
DefineScreen
MakeVScreen (TPWINDOW)

Declaration

```
procedure AssignCrt(var F : Text);
```

Purpose

Used to take over standard output by altering Output's FIB (File Interface Block).

Comments

This routine is intended almost exclusively for internal use. Like the AssignCrt routines in TPCRT and the standard CRT unit, it is used to reset the pointers in Output to point to TPSCREEN's own driver. Unlike the other two routines, however, it is not used to take over standard input (by altering Input).

See Also

InitVideo

Declaration

```
procedure CenterWrite(S : string);
```

Purpose

Centers a string on the current line in the active window.

Comments

This routine acts as a "shell" around SpeedWrite. It positions the cursor on the current row such that, after it calls SpeedWrite, S will appear centered. After the string is written, the cursor will be moved to the end of it.

Example

```
SelectScreen(0);
GotoXY(1,1);
CenterWrite('This string is centered.');
```

See Also

FastCenter (TPCRT) SpeedWrite

Declaration

```
procedure ColorColors;
```

Purpose

Switches the color map to color.

Comments

Intended primarily for internal use, this routine selects the default color map, rather than the monochrome color map described briefly in the documentation for MonoColors. Since TPSCREEN automatically selects the appropriate color map based on the current video mode, this routine should probably be called only while testing your program.

See Also

MonoColors

CopyScreen

Declaration

```
procedure CopyScreen(Handle : Byte);
```

Purpose

Copies the entire contents of a screen buffer to the active screen.

Comments

CopyScreen is very similar to CopyWindow, but it does not pay any attention to window coordinates. Instead, it attempts to copy the entire contents of the specified screen's buffer into the buffer for the active screen.

If the active screen does not have the same dimensions (the same number of rows and columns) as the screen specified by Handle (the source screen), CopyScreen will attempt to resize the active screen. This operation will succeed only if the active screen's buffer is as large or larger than the source screen's buffer.

For example, suppose the active screen is 10 columns wide and 4 rows high, and the source screen has 6 columns and 6 rows. A screen buffer is 2 * (Rows * Cols) bytes in size, so the active screen's buffer can hold 80 bytes (2 * (4 * 10)) and the source screen's buffer contains only 72 bytes (2 * (6 * 6)). Since the active screen is larger than the source, CopyScreen would simply change the active screen's dimensions to 6 columns by 6 rows (but not the size of the buffer itself), then copy the data.

If two screens have the same dimensions, no resizing is done, and the source screen's contents are immediately copied into the active screen.

Note that, besides copying the screen itself, CopyScreen will also position the cursor at the same relative location it was in in the inactive window.

Example

```
var
  I : Word;
begin
  ClrScr;
  for I := 1 to 24 do begin
    CenterWrite('Hello world!'); WriteLn;
  end;
  Write('Press any key to continue...');
  I := ReadKeyWord;
  Write(^M); ClrEol;
  if not AllocateScreen(1, CurrentCols, CurrentRows) then Halt;
  SelectScreen(1);
  CopyScreen(0);   {copy contents of Screen 0 into Screen 1}
  SelectScreen(0);
  ClrScr;
  GotoXY(1, 25);
  Write('Press any key to restore screen...');
  I := ReadKeyWord;
  CopyScreen(1);   {copy contents of Screen 1 into Screen 0}
end;
```

See Also

CopyWindow

MoveVScreenToWindow (TPWINDOW)

Declaration

```
procedure CopyWindow(Handle : Word);
```

Purpose

Copies a window from the specified screen to the active screen.

Comments

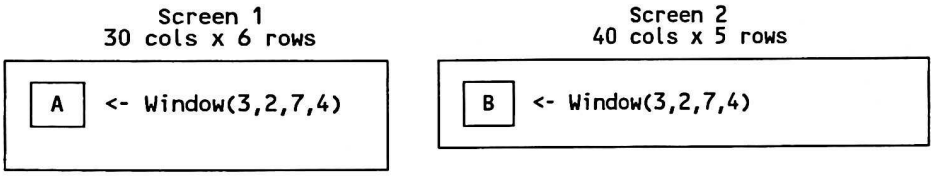
Like CopyScreen, this routine copies data from another screen's buffer to the active screen. However, CopyWindow *does* pay attention to window coordinates. The best way to understand what it does is to *see* what it does, so let's consider the following cases.

Case 1: The two screens have the same dimensions, and the active window is the size of the entire screen. In this case, CopyWindow(*n*) will behave exactly as CopyScreen(*n*) would.

Case 2: Screen 1, the active screen, was defined as having a *maximum* of 30 columns and 6 rows. Screen 2 was defined as having 40 columns and 5 rows. For Screen 1, the current window coordinates were set with 'Window(3,2,7,4);' (notice the window marked 'A'). Now let's say that we want to fill this tiny window with text from Screen 2. All that's required is

```
CopyWindow(2);
```

The result is that the contents of the small window marked B would be copied into the window marked A. (See the illustration below.)



Case 3: Same as Case 2, except that the window coordinates for Screen 1 were set with 'Window(1,1,30,6);' (the full screen). In this case, since Screen 2 is wider than Screen 1, only the first 30 columns (and all 5 rows) of Screen 2 would be moved, filling all but the bottom row of Screen 1.

Note that, besides copying the window itself, CopyWindow will also try to position the cursor at the same relative location that it was in in the inactive window.

CopyWindow

Example

```
var
  I : Word;
begin
  ClrScr;
  if not AllocateScreen(1, 30, 6) then Halt;
  if not AllocateScreen(2, 40, 5) then Halt;
  SelectScreen(2);
  Window(3,2,7,4);
  FillWindow('2');
  SelectScreen(1);
  Window(3,2,7,4);
  FillWindow('1');
  SelectScreen(0);  {copy to physical screen so we can see results}
  GotoXY(1, 25);
  Write('Press any key to see contents of screen 1...');
  I := ReadKeyWord;
  CopyWindow(1);    {copy contents of screen 1 into screen 0}
  GotoXY(1, 25);
  Write(^M'Press any key to see contents of screen 2...');
  I := ReadKeyWord;
  CopyWindow(2);    {copy contents of screen 2 into screen 0}
  GotoXY(1, 25);
end;
```

This example is similar to Case 2, above, except that the data is copied onto the physical screen so we can see the results.

See Also

CopyScreen

Declaration

```
function CurrentCols : Byte;
```

Purpose

Returns the number of columns in the active window.

Comments

This function returns the width of the active window.

See Also

CurrentRows

VScreenCols (TPWINDOW)

Declaration

```
function CurrentRows : Byte;
```

Purpose

Returns the number of rows in the active window.

Comments

This function returns the height of the active window.

See Also

CurrentCols

VScreenRows (TPWINDOW)

DeallocateScreen / DefineScreen

Declaration

```
procedure DeallocateScreen(Handle : Byte);
```

Purpose

Deallocates a screen buffer allocated by AllocateScreen.

Comments

This routine won't deallocate a buffer not allocated by AllocateScreen. If no buffer has been allocated, it does nothing. Handle must be in the range 1..MaxScreens.

Examples

```
DeallocateScreen(0);
```

Has no effect because 0 is the handle for the physical screen.

```
DeallocateScreen(1);
```

Deallocates the buffer for window 1.

See Also

AllocateScreen

DisposeVScreen (TPWINDOW)

Declaration

```
procedure DefineScreen(BufAddr : Pointer; Handle, Cols, Rows : Byte);
```

Purpose

Defines a virtual screen.

Comments

This is a low-level routine that you should not need to use unless you want to manage screen buffers yourself. It differs from AllocateScreen only in that it assumes that you have already allocated a buffer (pointed to by BufAddr) and initialized it.

Handle is the number of the screen to be defined, and should be in the range 1..MaxScreens. Cols and Rows specify the dimensions of the screen.

Examples

```
DefineScreen(MyBufferPointer^, 1, CurrentCols, CurrentRows);
```

Incorrect! The '^' in 'MyBufferPointer^' is unnecessary.

```
DefineScreen(MyBufferPointer, 1, CurrentCols, CurrentRows);
```

Correct. Defines Screen 1 as having the same dimensions as the active screen. MyBufferPointer points to the screen's buffer.

See Also

AllocateScreen

MakeVScreen (TPWINDOW)

Declaration

```
function FastReadLine(Handle, Row : Byte) : string;
```

Purpose

Reads a complete line of text from any screen and returns it as the function result.

Comments

This routine is similar to the FastRead routine in TPCRT, but it is intended for reading complete lines and, unlike FastRead, it works with virtual screens as well as the physical screen. Given a screen (Handle) and row number (Row), it will return the entire text of the line, trimmed of trailing spaces.

Example

```
SelectScreen(0);
WriteLn('HelloWorld!');
WriteLn(FastReadLine(0, 1));
```

This would display 'HelloWorld!' twice.

See Also

FastRead (TPCRT)

ReadVScreen (TPWINDOW)

Declaration

```
procedure FastWriteMem(St : string; var Address; Attr : Byte);
```

Purpose

Write St at Address in Attr (video attribute) without snow or flicker.

Comments

Intended primarily for internal use (by SpeedWrite, for example), this routine will write a string to any location in memory, using Attr as the video attribute. If Address is a region of video memory, snow and flicker are prevented. Internally, this routine is pretty much identical to FastWrite (TPCRT)--which can theoretically write to any location in memory--except that it does not calculate where the string is to be written.

Example

```
FastWriteMem('Hello world!', Mem[VideoSegment:0], TextAttr);
```

Write 'Hello world!' at the top of the screen using the default video attribute.

See Also

FastWrite (TPCRT)

FillAttribChar / FillVideoChar

Declaration

```
procedure FillAttribChar(var Dest; Number : Word; Ch : Char; Attr : Byte);
```

Purpose

Fills Dest with Number character/attribute pairs.

Comments

Intended primarily for internal use (by FillWindow and FillVideoChar), this routine will fill any contiguous region of memory with the specified character and video attribute. If Dest is a region of video memory, snow and flicker are prevented.

Example

```
FillAttribChar(Mem[VideoSegment:0], CurrentRows*CurrentCols, '$', TextAttr);
```

Fill the entire (physical) screen with dollar signs using the default video attribute.

See Also

FillVideoChar FillWindow

Declaration

```
procedure FillVideoChar(R, C : Byte; Number : Word; Ch : Char; Attr : Byte);
```

Purpose

Fills Number character/attribute pairs at row R, column C.

Comments

Intended primarily for internal use (by TPSCREEN's ClrEol), this routine will fill any contiguous portion of a screen with the specified character and video attribute.

Note: Although the routine checks to see if the starting point is inside the active window, it does not verify that Number character/attribute pairs will fit in the window.

Example

```
SelectScreen(0);
FillVideoChar(1, 1, CurrentCols, '*', TextAttr);
```

Fill the line at the top of the screen with asterisks, using the default video attribute. This is equivalent to the example given for FillWindow.

See Also

FastFill (TPCRT) FillWindow

Declaration

```
procedure FillWindow(Ch : Char);
```

Purpose

Fills the active window with the specified character.

Comments

Intended primarily for internal use (by TPSCREEN's ClrScr), this routine will fill the entire window with the specified character. It uses the current video attribute (TextAttr), which can be altered directly ('TextAttr := $07') or indirectly (using TextColor and TextBackground).

Example

```
SelectScreen(0);
Window(1, 1, 80, 1);
FillWindow('*');
```

Create a one-line window at the top of the physical screen and fill it with asterisks.

See Also

ClearVScreen (TPWINDOW) FillVideoChar

Declaration

```
procedure InitVideo;
```

Purpose

Initializes TPSCREEN's data structures and variables.

Comments

Since TPSCREEN is self-initializing, this routine should rarely need to be called, if ever. The only kind of case where you might need it would be one in which you called ReleaseVideo to temporarily disable TPSCREEN's video routines, and wished to reenable them later. Calling InitVideo has no effect if initialization has already been done.

See Also

ReleaseVideo

MonoColors

Declaration

```
procedure MonoColors;
```

Purpose

Switches the color map to monochrome/black and white.

Comments

Intended primarily for internal use, this routine selects the monochrome color map, so that colors selected for a color monitor will look more tolerable on a monochrome or composite black-and-white monitor. For example, after MonoColors has been called

```
TextColor(Yellow);
TextBackground(Black);
```

would have the same effect as

```
TextColor(White);
TextBackground(Black);
```

Since TPSCREEN automatically selects the appropriate color map based on the current video mode, this routine should probably be called only while testing your program, so that you can see the effects of color mapping.

See Also

ColorColors

Declaration

```
procedure MoveWindowHorizontal(Handle, Cols : Byte; MoveRight : Boolean);
```

Purpose

Moves the active window Cols columns to the left/right, filling in the background with text from the specified screen.

Comments

This routine moves the contents of the active window left (MoveRight = False) or right (MoveRight = True) by the specified number of columns (Cols). Like MoveWindowVertical, it uses text from another screen (specified by Handle) as filler, rather than creating blank columns. The screen used as background *must* have the same dimensions (number of rows and columns) as the active screen, or the routine will do nothing.

Let's say that the window for the active screen (0) is a full 80 columns by 25 rows, that the screen is empty except for the first column, which is filled from top to bottom with the letter '0', and that Screen 2 is empty except for the last column, which is similarly filled from top to bottom with the letter '2'. If we executed

```
MoveWindowHorizontal(2, 1, False);
```

then the column filled with '0' would be moved off Screen 0 (permanently) and the column filled with '2' would appear at the far right edge of it. (See the example below.)

Example

```
var
  I : Word;
  Ch : Char;
begin
  ClrScr;
  for I := 1 to 24 do
    WriteLn('0');
  if not AllocateScreen(2, 80, 25) then Halt;
  SelectScreen(2);
  ClrScr;
  for I := 1 to 24 do begin
    GotoXY(80, I);
    Write('2');
  end;
  SelectScreen(0);
  Write('Press any key...');
  Ch := ReadKey;
  MoveWindowHorizontal(2, 1, False);
end;
```

See Also

MoveWindow (TPWINDOW) MoveWindowVertical

MoveWindowVertical

Declaration

```
procedure MoveWindowVertical(Handle, Lines : Byte; MoveDown : Boolean);
```

Purpose

Moves the active window Lines rows up or down, filling in the background with text from the specified screen.

Comments

This routine moves the contents of the active window up (MoveDown = False) or down (MoveDown = True) by the specified number of Lines. Unlike ScrollWindowUp and ScrollWindowDown, however, it uses lines from another screen (specified by Handle) as filler, rather than creating blank lines, and it can be used on virtual screens as well as the physical screen. The screen used as background *must* have the same dimensions (number of rows and columns) as the active screen, or the routine will do nothing.

Let's say that the window for the active screen (0) is a full 80 columns by 25 rows, that the screen is empty except for the first line, which says 'Line 1, Screen 0', and that Screen 2 is empty except for the last line, which says 'Line 25, Screen 2'. If we executed

```
MoveWindowVertical(2, 1, False);
```

then 'Line 1, Screen 0' would be moved off the screen (permanently) and 'Line 25, Screen 2' would appear at the bottom of the screen. (See example below.)

Example

```
var
  Ch : Char;
begin
  ClrScr;
  WriteLn('Line 1, Screen 0');
  if not AllocateScreen(2, 80, 25) then Halt;
  SelectScreen(2);
  ClrScr;
  GotoXY(1, 25);
  Write('Line 25, Screen 2');
  SelectScreen(0);
  Write('Press any key...');
  Ch := ReadKey;
  MoveWindowVertical(2, 1, False);
end;
```

See Also

MoveWindow (TPWINDOW)
MoveWindowHorizontal
ScrollWindowUp
ScrollWindowDown

Declaration

```
procedure ReinitScreen;
```

Purpose

Reinitialize the screen.

Comments

This routine is TPSCREEN's equivalent of TPCRT's ReinitCrt, which it in fact calls. ReinitScreen reinitializes all global variables pertaining to video mode, screen size, and screen type. It automatically selects the physical screen as the active screen, the visual page as the active page, and the full screen as the active window.

ReinitScreen *must* be called after every mode change before trying to write to the physical screen. If you use Exec (DOS) or ExecDos (TPDOS) to execute another program, keep in mind that the program you execute can change the video mode. This means that ReinitScreen should probably be called after each call to Exec or ExecDos. Similarly, if you are using TPSCREEN in a TSR, you will usually need to call ReinitScreen each time your popup is invoked.

Example

```
if CurrentMode <> 7 then begin
  TextMode(1);
  ReinitScreen; {!}
  WriteLn('This is 40-column mode. Press any key...');
  Ch := ReadKey;
  TextMode(3);
  ReinitScreen; {!}
  WriteLn('This is 80-column mode.');
end
else
  WriteLn('Requires color adapter');
```

Notice that ReinitScreen is called after each call to TextMode.

See Also

ReinitCrt (TPCRT)

SelectScreen

ReleaseVideo

Declaration

```
procedure ReleaseVideo;
```

Purpose

Routes all output through TPCRT.

Comments

This routine disables most of TPSCREEN's display routines, thereby routing screen output through TPCRT. The most typical use for it is to prevent error messages displayed by an exit handler from being written to a virtual screen rather than the physical screen. Although TPSCREEN calls ReleaseVideo automatically when a program ends, to allow any messages generated by Turbo's exit handler to get through, this call will typically not be made until after your exit handler(s) have been executed.

Example

```
var
  SaveExitProc : Pointer;
...
{$F+}
procedure MyExitProc;
begin
  ExitProc := SaveExitProc;
  ReleaseVideo;
  WriteLn('Error ', ExitCode, ' at ', HexW(Seg(ErrorAddr^)), ':',
    HexW(Ofs(ErrorAddr^)));
end;
{$F-}
...
SaveExitProc := ExitProc;
ExitProc := @MyExitProc;
```

See Also

InitVideo

RestoreWindowCoordinates / SelectScreen

Declaration

```
procedure RestoreWindowCoordinates(WC : WindowCoordinates);
```

Purpose

Restore previously saved window coordinates.

Comments

This routine allows you to restore window coordinates previously saved using SaveWindowCoordinates. Note that the same screen should usually be active when you restore the window coordinates that was active when you saved them.

See the entry for StoreWindowCoordinates for an example.

See Also

EraseWindow (TPWINDOW) StoreWindowCoordinates
RestoreWindowCoordinates (TPCRT)

Declaration

```
procedure SelectScreen(Handle : Byte);
```

Purpose

Routes screen writes to the specified window.

Comments

This routine lets you specify where subsequent screen writes will go. If Handle = 0, it routes output to the physical screen, otherwise to a virtual screen. To display a virtual screen (call it ScreenNum), you must (1) make the physical screen the active screen using 'SelectScreen(0);', then (2) copy the contents of the virtual screen to the physical screen using 'CopyScreen(ScreenNum);'.

Example

```
var I : Word;
begin
  if not AllocateScreen(1, CurrentCols, CurrentRows) then Halt;
  SelectScreen(1); ClrScr;
  for I := 1 to CurrentRows do begin
    CenterWrite('Hello world!');  WriteLn;
  end;
  SelectScreen(0);
  Write('Press any key...'); I := ReadKeyWord;
  CopyScreen(1);
end;
```

See Also

ActivateVScreen (TPWINDOW) ReinitScreen
CopyScreen

SetActivePage / SetBackgroundChar

Declaration

```
procedure SetActivePage(PageNo : Byte);
```

Purpose

Sets the video page that will be written to with subsequent operations on the screen.

Comments

As explained in the overview for this section, the *active page* is the video page currently being written to, as opposed to the *visible page*, which is the one currently being displayed. SetActivePage has no effect if a color adapter is not in use, since monochrome adapters have only one video page.

SetActivePage does not select the physical screen if it isn't selected already. To do that, you must use 'SelectScreen(0);'.

Example

```
SetActivePage(1);
ClrScr;
WriteLn('Hello world!');
SetVisiblePage(1);
```

Make page 1 the active page, clear the screen, write a string, then make page 1 the visible page so you can see what was written.

See Also

SetVisiblePage (TPCRT)

Declaration

```
procedure SetBackgroundChar(Ch : Char);
```

Purpose

Sets the background character to be used by ClrScr and ClrEol.

Comments

Normally ClrScr and ClrEol use a space (' ') when filling the screen. This routine allows you to select an alternate character to be used as filler.

Example

```
SetBackgroundChar(#177);
ClrScr;
```

Fill the active screen with character #177.

See Also

TextChar (TPCRT)

Declaration

```
procedure SpeedWrite(S : string);
```

Purpose

Writes a string onto the screen at the current cursor location with the current attribute.

Comments

SpeedWrite blends the simplicity of a Pascal Write statement with the speed of TPCRT's FastWrite routine, and (unlike FastWrite) it can be used to write to virtual screens. After writing the specified string to the screen at the current position of the cursor, SpeedWrite moves the cursor to the end of the string, just as a Write statement would.

A couple of caveats. First, like FastWrite, SpeedWrite displays control characters such as carriage returns and line feeds, so you cannot use it in place of WriteLn. Second, if the entire string will not fit on the current line, SpeedWrite will truncate it. For example, if the string is 81 characters in length, the window is 80 characters wide, and the cursor is in column 1, the last character in the string will not be displayed. Finally, if the cursor is outside the boundaries of the active window, SpeedWrite will do nothing.

Examples

```
SpeedWrite('Hello world!'^M^J);
```

Probably not what you'd want. The ^M (carriage return) and ^J (line feed) would be displayed as a musical note and a box with a circle in it.

```
SpeedWrite('Hello world!');
WriteLn;
```

This is probably what you'd want.

```
SpeedWrite(String1, String2);
```

Incorrect! Unlike Write, you can pass only one string parameter to SpeedWrite at a time.

```
SpeedWrite(String1 + String2);
```

Correct.

See Also

FastWrite (TPCRT)
FastWriteClip (TPWINDOW)
FastWriteMem
WriteVScreen (TPWINDOW)

StoreWindowCoordinates

Declaration

```
procedure StoreWindowCoordinates(var WC : WindowCoordinates);
```

Purpose

Store the current window coordinates.

Comments

This routine allows you to save the window coordinates for the active window. They can later be restored using RestoreWindowCoordinates. Note that the same screen should usually be active when you restore the window coordinates that was active when you saved them.

Example

```
var
  WC : WindowCoordinates;
begin
  StoreWindowCoordinates(WC);
  Window(1, 1, 80, 10);
  {-- something --}
  RestoreWindowCoordinates(WC);
end;
```

See Also

StoreWindowCoordinates (TPCRT) RestoreWindowCoordinates

3. Data Entry

A. Overview

The TPENTRY unit provides a powerful full-screen data entry facility. Unlike the routines in TPEDIT, which are designed for editing individual strings and numbers, TPENTRY allows you to group multiple fields together in a single entry screen. You lay out the screen, and TPENTRY handles the tasks of editing, cursor movement, range checking, and validation of input.

TPENTRY supports the following basic data types:

- Character: Char, String, Array of Char
- Scalar: Byte, ShortInt, Word, Integer, LongInt
- Floating point: Real, Single, Double, Extended, Comp, BCD
- Logical: Boolean, Yes/No
- Date and time: Julian date, date string, time

TPENTRY also, however, provides hooks that let you adapt it to deal with enumerated types and other user-defined data structures.

Validation of input is handled in two phases. Picture masks allow limited validation during the editing of a field, while built-in or user-written routines may be used to perform range checking and other, more comprehensive forms of validation after editing is complete.

Other notable features of TPENTRY include:

- An extensive list of built-in commands for editing and movement among fields, as well as hooks for adding up to 10 new commands
- Redefinable keyboard
- Extensive mouse support
- Required and protected fields
- Hooks for implementing calculated fields
- Multiple choice fields
- Optional right-to-left (calculator-style) editing of numeric fields
- Horizontal scrolling within edit fields
- Data entry screens of up to 80 columns and 127 rows; display is scrolled vertically as necessary to accommodate forms longer than the physical screen
- Hook for implementing nested data entry screens
- Read-only mode to allow browsing through records without editing them

- Password mode for suppressing the display of text being entered
- Discretionary commas and floating/fixed currency symbols for numeric fields
- Complete programmer control over the display of help and error messages
- Hooks for implementing background tasks and context-sensitive help
- Facilities for altering the display of date, time, and numeric fields to account for differing monetary units and conventions
- Numerous options to control the behavior of the field editors and the entry screen manager
- Optional color mapping for monochrome displays

TPENTRY's companion unit, TPDATE, provides routines for working with Julian dates, date strings, and time variables. A Julian date is represented as a long integer that contains the number of days since January 1, 1600 and can store any valid date through December 31, 3999. This method of storing dates has the advantages of being compact and making date arithmetic simple. A date string is a string representation of a date, and may be in any format you desire: MM/DD/YY, DD/MM/YY, MM/DD/YYYY, etc. As with dates, times are represented as long integers containing the number of seconds since midnight, again facilitating arithmetic manipulation of them.

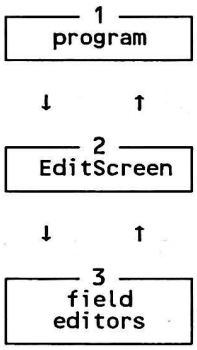
B. TPENTRY: Data Entry Screens

When you see this section of the manual for the first time, you may be tempted to conclude that TPENTRY is tòo large and complex for you to master. But don't panic. Although this is the largest unit in the package, TPENTRY is not really all that complex. The size of the unit's interface section is due primarily to its wealth of options, many of which you'll never use and most of which are readily understood. The example program ENTRY.PAS demonstrates how to use most of them.

How TPENTRY Works

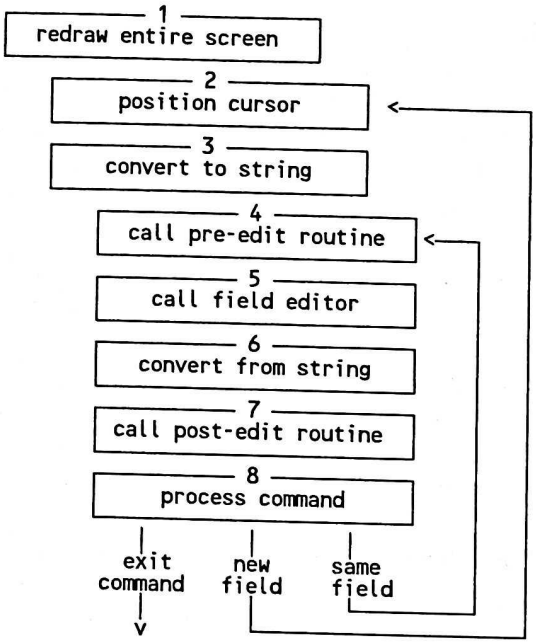
The following is a broad overview of how the whole data entry system works. It is our hope that, given a general understanding of how control flows from one part of the system to another, you will be better able to understand not only how to use the system, but also how to take advantage of the numerous hooks it makes available.

From the broadest view, control flows like this:



That is, the main program calls EditScreen to handle the editing of all the fields in the entry screen. It in turn calls upon one of the "field editors" (see "The Three Field Editors" later in this section) to process the individual fields. When the field editor returns control, EditScreen either moves the cursor to another field and continues editing, or returns control to the main program. Depending on the "exit command" issued by the user (see "Example" later in this section), the program may either terminate the editing session or make another call to EditScreen. (For an example of how to process the various exit commands, see ENTRY.PAS.)

Sharpening the focus a bit, the flow of control within EditScreen itself looks something like this:



On entry to EditScreen, the entire screen is drawn (box 1). (We'll skip the details of how that's done because you don't really need to know.) After that, control flows into the larger of the routine's two main loops.

At the beginning of this loop (comprising boxes 2-8), the cursor is moved into the current field, then the "conversion routine" for that field is called (see "Conversion Routines" later in this section). Its job in this situation is to convert the variable associated with the current field from its native type (integer, longint, real, date, etc.) into a string, then merge that string with the field's "picture mask" (see "Picture Masks" later in this section). This allows the field editor to perform its duties without any real knowledge of the field's type.

Digressing for a moment: For each data type supported by TPENTRY there are three corresponding routines: one to add a field of that type to the entry screen, one to perform conversions, and one to perform validation. This "modular" approach offers two distinct advantages: (1) Turbo Pascal's smart linker can strip out all the code that handles an unused data type, and (2) the programmer can expand TPENTRY to handle new data types simply by writing the necessary conversion and validation routines. (See "User-defined Types" later in this section.) Now back to our flow charts...

Control then flows into the innermost loop (boxes 4-8). The first task here is to call the "pre-edit routine," if there is one (see "Displaying Help Messages" later in this

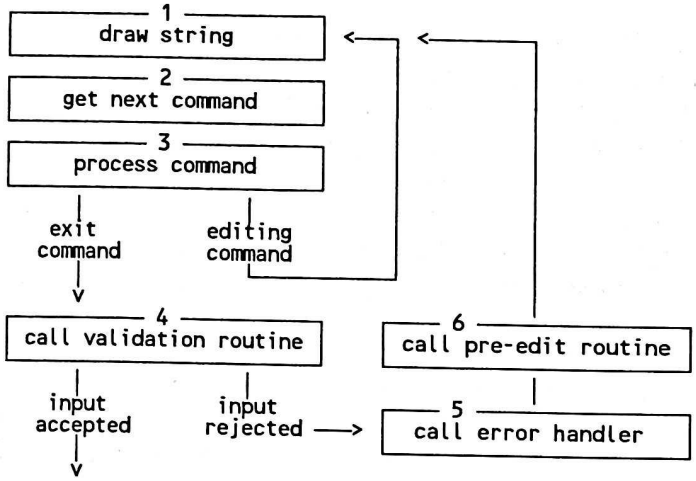
section). This gives the calling program a chance to display a help message if desired. The next step is to call one of the three field editors: the "string editor," the "numeric editor," or the "multiple-choice editor". We'll discuss the job of the field editors in more detail later. When the field editor returns control to EditScreen, the conversion routine is called again, this time to convert the string being edited back to its native type. This process requires two steps: one to strip out the picture mask characters, and a second to convert the raw string into the appropriate type. (For examples of conversion routines, see TPENTRY.IN1.)

The next step is to call the "post-edit routine," if there is one (see "Calculated Fields" later in this section). This is unquestionably the most important hook provided to the programmer by TPENTRY, and understanding it is the key to mastering the power of the system. You can do many things in a post-edit routine: update a calculated field; protect or unprotect a field; hide or unhide a field; change the command entered by the user; force the cursor to move to a particular field; the list goes on. The more common possibilities are discussed in this section, and most of them are demonstrated in ENTRY.PAS. But the possibilities are almost unlimited, as some of our existing customers have proven.

After the post-edit routine is called, EditScreen is ready to evaluate the command that caused the field editor to return control. This command usually falls into one of two categories: an "exit command" (such as <Esc> or <CtrlEnter>) or a cursor movement command (such as <Enter>, <Up>, or <Down>). If it is an exit command, EditScreen returns control to the calling program, passing back the code for the exit command as its function result. If it is a cursor movement command, EditScreen has to calculate the field to move to. In some cases, the cursor may not need to be moved at all--for example, if the current "wrap mode" (see SetWrapMode) is StopAtEdges, and the user tries to move the cursor beyond the last row by pressing <Down>--and control flows back to the top of the inner loop (box 4). In most cases, however, the cursor will be moved to another field, and control flows back to the top of the outer loop (box 2).

Digressing again for a moment: The category "exit commands" itself comprises two types of commands, pre-defined commands and user-defined commands. Creating a user-defined exit command is a simple two-step process: first you call AddEntryCommand to assign your command to a key (see "Installable Keyboard Hooks" in Appendix B), then you write the code to process the command. For some good examples of how to use this hook, see ENTRY.PAS.

That pretty much covers the work done by EditScreen, so let's sharpen the focus a bit further and look at what the field editors do:



Each of the three field editors has a large main loop. At the top of the loop, the string being edited is drawn and the cursor is moved into position (box 1). The next step is to get a command from the user. Here again, you have an opportunity to gain temporary control. By changing the value of the global variable EntryKeyPtr, you can set up a customized keyboard input routine (which we'll call "GetKey"). The GetKey routine's main job is to return the next keystroke entered by the user, but it can do any number of things while it is waiting for that event, if it does them quickly enough. This facility is known as the "Background Task Hook," and it is discussed at length in Appendix B.

Once the keyboard handler returns the next key and it is translated into a command, the field editor must process that command. Here again there are two broad categories of commands: exit commands and editing commands. Editing commands are commands that modify the contents of the field and/or move the cursor from one part of it to another. The category of exit commands comprises both the "true" exit commands (those that are treated as exit commands by EditScreen) and those commands that cause the cursor to be moved from one field to another.

When an exit command is given, the field editor does one of two things. If <Esc> was pressed (corresponding to the ESquit command), it restores the original value of the string being edited and exits immediately. Otherwise, it calls the "validation routine" associated with the current field. (TPENTRY.IN1 and ENTRY.PAS contain examples of validation routines.) The task of the validation routine varies depending on the data type, but in general it must do two things: convert the string being edited back to its native type, then determine whether the value entered is acceptable. If the value is acceptable, it returns True, telling the field editor to return control to EditScreen. If the value is unacceptable, it returns False, passing back an error code (see the error codes in TPENTRY:Declarations) and a pointer to an error message.

The field editor then either calls the user-defined error routine if there is one (see "Error Handling" later in this section) or "rings the bell" if there isn't. (In most applications you will probably want to use of this hook, because the user will rarely understand what was wrong if you don't display the error message.) The field editor then calls the pre-edit routine (if there is one) again, in case the help message was overwritten by the error message, and goes back to the start of the loop (box 1). This process is repeated until a valid entry is made by the user or <Esc> is pressed.

That pretty much covers the major components of the data entry system provided in TPENTRY. It is a complex system, to be sure, but hopefully this overview will give you the confidence to explore it further. After you've worked with it a while, we think you'll agree that the system's uncommon flexibility is responsible for its complexity, and that ultimately the end justifies the means.

Example

A good way to become familiar with TPENTRY is to consider a simple example:

```
{$V-}   {<-- compiler directive required for use of TPENTRY}
program SimpleExample;
uses
  TpCrt, TpEntry;
var
  ESR : ESrecord;
  ExitCommand : EStype;
  MyRec : record
            Name : string[30];
            Phone : string[14];
          end;
begin
  ClrScr;
  FillChar(MyRec, SizeOf(MyRec), 0);
  InitESrecord(ESR);
  SetPromptAttr($0F);
  SetFieldAttr($70);
  SetStringAttr($70);
  SetCtrlAttr($70);
  AddStringField(
    ESR, 'Name:',  1, 1, '',                1, 8, 30, 0, nil, MyRec.Name);
  AddStringField(
    ESR, 'Phone:', 2, 1, '(999) 999-9999', 2, 8, 14, 0, nil, MyRec.Phone);
  ExitCommand := EditScreen(ESR, 0, False);
  DisposeEditScreen(ESR);
end.
```

Let's look at each section of the program in detail. In the declaration section, there are three variables: ESR contains the data used by the edit screen manager to keep track of all the fields that make up the entry screen; ExitCommand holds the return code passed back by the screen manager after all the fields have been edited; and MyRec holds the data that we want to edit.

The FillChar statement near the beginning of the program is simply used to initialize the data structures that we will be editing. This step is important because the field editors have no way of determining whether or not a given field already contains meaningful data.

InitESrecord must be called before any fields are added to an ESrecord to insure that it is initialized properly. To prevent simple oversights from causing system crashes, this routine inserts into the ESrecord a special "signature" to indicate that it has been initialized. Any attempt to add fields to an improperly initialized edit screen record will result in a fatal error, signalled by a short beep, causing the program to halt. This routine also sets the default coordinates for the data entry screen. See the section on "Scrollable Data Entry Screens" for more details.

The four procedure calls that follow establish the colors to be used for displaying prompts (SetPromptAttr), inactive fields (SetFieldAttr), the active field (SetStringAttr), and any control characters within the field being edited (SetCtrlAttr). In this case, prompts are displayed in white on black, and both active and inactive fields are displayed in reverse video. Although in this example these colors are set only once, they may be changed freely while fields are being added in order to achieve special effects. (You might, for example, want to display required, protected, or calculated fields in special colors.)

The calls to AddStringField establish the two fields that make up our simple data entry screen. Its parameters are as follows:

```
procedure AddStringField(
  var ESR : ESrecord;        {the edit screen to add to}
  Prompt : string;           {edit prompt}
  pRow, pCol : Byte;         {coordinates for prompt}
  Picture : string;          {edit mask}
  eRow, eCol,                {coordinates for edit field}
  EditLen : Byte;            {width of edit field}
  HelpIndex : Word;          {index to pass to help routine}
  Validation : Pointer;      {pointer to special validation routine}
  var EditSt : string);      {string to edit}
```

Among other things, these parameters specify the text and location for the prompt associated with the field; the picture mask for the field, if any; the location and width of the edit field; and the string variable to be edited. In the first case (Name), there is no picture mask. This indicates to TPENTRY that the string may contain any character; the width of the edit field (30) is therefore used to determine how long the string can be. In the second case (Phone), both the length of the picture mask and the width of the edit field are 14. If the Picture mask were longer than the width of the edit field, the field editor would display only part of the string being edited at any one time, and would scroll horizontally as necessary.

In most cases, the screen coordinates passed to AddStringField (and other such routines) are treated as absolute, meaning that Row 1, Col 1 refers to the top left corner of the screen. If the SetEntryWindow routine has been called, however, the coordinates will be interpreted as relative to the specified window. Again, see the section on "Scrollable Data Entry Screens" for more details.

We should probably point out here that AddStringField is only one of many routines available for adding fields to an entry screen. Similar routines exist for adding fields of other types: AddByteField (Byte), AddWordField (Word), AddRealField (Real), AddDateField (Julian dates), and so on.

A few more points about adding fields:

First, it is assumed that fields will be added in a specific order based on their location on screen: left to right, and top to bottom. This is necessary to allow the screen manager to move the cursor up, down, left, or right upon request. Imagine, for example, a screen consisting of nine fields arranged in three rows and three columns, with the cursor sitting in the middle field (second row, second column). To move to the field in the middle of the top row, you would simply press the <Up> arrow key; to move to the middle of the bottom row you would press <Down>. Similarly, to move to the left- or rightmost columns, you would press <Left> or <ShiftTab>, or <Right> or <Tab> depending on the current location of the cursor. In contrast to other data entry screen managers, which generally have no concept of space, TPENTRY can thus provide a logical interface that allows the user to move from field to field in an intuitive fashion.

Second, as each field is added it is automatically assigned a "field ID" (a number between 0 and 254). The first field added is assigned an ID of 0, the second one is assigned an ID of 1, and so on. This value uniquely identifies each field in an entry screen. Among other things, it allows the screen manager to tell user-written routines which field is currently being edited.

Once all the fields have been added, we can begin editing by calling EditScreen, which takes three parameters. The first is the name of the edit screen record. The second is the ID of the first field to be edited. Normally you will want to specify 0 here, but in other cases you may want or need to specify an alternate value. For example, if the topmost field on screen is an unprotected field that rarely needs to be changed, you might want to pass a parameter of 1, to position the cursor initially on the second field. The third is used to select read-only mode, in which the user can move around the data entry screen, but cannot edit any of the fields. This option is especially useful when working with data entry screens larger than the physical screen, but in most cases this parameter will be False.

As indicated earlier, EditScreen automatically handles all requests to move from one field to another. However, several built-in editing commands--referred to collectively as "exit commands"--can indicate that the user wants to edit a different record, cancel all edits made to the current record, or to end an editing session entirely. (You can also define your own exit commands.) The exit command that was used in a given case is returned by EditScreen as its function result. In our example program, this command is stored in ExitCommand, but is never acted upon. In a real application, you will generally need to write code to handle all possible exit commands, including any that you've defined yourself. See ENTRY.PAS for an example of how to do this.

The final line in the example program, the call to DisposeEditScreen, is actually unnecessary in this case. It disposes of all the memory that was allocated dynamically when fields were being added to the edit screen, then calls InitESrecord to reinitialize the ESrecord. Since our example program ends here, there is no need to recover the memory that was allocated. In large applications, however, it will often be desirable to recover this memory after each editing session. See the section on DisposeEditScreen for more details.

So, just to recap: The only two procedures that you *must* call when using TPENTRY for data entry are InitESrecord and EditScreen. In almost all cases, you will want to call the four routines (SetPromptAttr, SetFieldAttr, SetStringAttr, and SetCtrlAttr) that determine the colors that fields are displayed in. And you will also need to call one or more of the routines that, like AddStringField, add an edit field to a data entry screen. Everything else is bells and whistles, which you can use or ignore as you see fit.

Now that you have a general idea of what's involved in using TPENTRY, we can back up and cover some basic concepts, starting with picture masks.

Picture Masks

The picture mask for a field tells TPENTRY not only how data is to be formatted during editing, but also what kinds of characters may be entered in particular positions. A picture mask may consist of one or more of three kinds of characters: formatting characters, literal characters, and semi-literal characters.

Formatting characters denote those places in a field where new text can be entered by the user or existing text can be modified. Each formatting character is associated with its own character set, and only characters within that set may be entered in the corresponding slot. For example, the picture mask '999' indicates that only numbers ('0'..'9') and spaces may be entered in a given field. Several formatting characters--'!', 'A', and 'L', for example--also indicate that input should automatically be converted to upper or lower case.

In general, any character in a picture mask that has not been assigned a function as a formatting character is treated as a literal, meaning that that location in the field cannot be altered during editing. In the case of fields of type string, these literal characters become a permanent part of the string being edited, to insure that they will look the same way when you display them yourself as they do when they are being edited. In most other cases, the literal characters in a picture mask are stripped out in the process of converting a string back to its original data type, and restored when converting it back to a string.

Semi-literal characters are like literal characters in the sense that they usually correspond to locations in a field that cannot be altered while editing. The two currency characters ('c' and 'C'), for example, always mark slots in a field that cannot be changed. What appears on the screen and the edited string, however, is usually a '$' (this can be changed).

In a couple of cases, however, a semi-literal character requires more than a simple substitution. The floating dollar sign ('$') sets aside a slot in a numeric field that will be used only if the rest of the field is full. If the field is not full, the dollar sign will be moved just to the left of the leftmost digit. Somewhat similarly, the comma character (',') indicates a slot that is to be filled with a comma (or the value of CommaChar) only if there is a digit to put to the left of it. Otherwise its slot is filled with a dollar sign, minus sign, or space. In neither case, however, can the slot set aside for the semi-literal characters be filled by a digit entered by the user.

Before this gets too abstract, let's look at the formatting characters defined by TPENTRY, some of which will be familiar to dBase programmers:

Char	TPENTRY accepts
'X'	#0..#255 (any character)
'!'	#0..#255 (forced to upper case)
'L'	#0..#255 (forced to lower case)
'a'	'A'..'Z', 'a'..'z', #128..#154, #160..#167, ' ', '-', '.', ',' (alphas only)
'A'	(same as 'a', but forces upper case)
'l'	(same as 'a', but forces lower case)
'B'	'T', 'F' (boolean True/False, forced to upper case)
'Y'	'Y', 'N' (Yes/No, forced to upper case)
'9'	'0'..'9', ' ' (unsigned numbers only)
'#'	'0'..'9', ' ', '-', '.' (signed or unsigned numbers)
'1'	corresponds to user-defined character set (UserSet1); optionally forced to upper or lower case based on value of ForceCaseUser['1']
'2'..'8'	correspond to UserSet2..UserSet8, ForceCaseUser['2'..'8']

And these are the semi-literal characters:

Char	TPENTRY converts to
'c'	'$' or CurrencyLtStr (fixed currency symbol–left)
'C'	' ' or CurrencyRtStr (fixed currency symbol–right)
'$'	'$' or CurrencyLtStr (floating currency symbol–left)
'.'	'.' or DecimalChar (fixed)
','	',' or CommaChar (discretionary)

Important note: The semi-literal characters ',' and '$' are allowed only in fields containing fixed decimal points and in numeric fields in which there can be no decimal point (integer fields, real fields with a decimal places setting of 0, etc).

In addition, TPENTRY also recognizes the following formatting and semi-literal characters, which are defined in the companion unit TPDATE:

Character	TPENTRY accepts or converts to
'm','d','y'	'0'..'9', ' ' (for date fields)
'M','D'	(same)
'/'	'/' or SlashChar (fixed, semi-literal)
'h','m','s'	'0'..'9', ' ' (for time fields)
'H','M','S'	(same)
':'	':' or ColonChar (fixed, semi-literal)
't'	'p', 'P', 'a', 'A' (case depends on UpCaseTime)
'e'	'm' or 'M' ('e' as in 'em', case depends on UpCaseTime)

To make the use of these formatting characters and the distinctions among them clearer, here are some examples of some typical picture masks and some resulting strings:

Picture	Result	Remarks
'XXXXXXXXXXXX'	'Hello world!'	Any character accepted
'!!!!!!!!!!!!'	'HELLO WORLD!'	Any character, force upper case
'LLLLLLLLLLLL'	'hello world!'	Any character, force lower case
'AAAAAAAAAAAA'	'HELLO WORLD '	alphas only, force upper case
'llllllllllll'	'hello world '	alphas only, force lower case
'[9999]'	'[123]'	digits and ' ' only, brackets are literal
'[####]'	'[-123]'	'-' is allowed
'c999.99'	'$ 12.50	dollar sign is fixed
'$999.99'	' $12.50'	dollar sign floats
'999.99CC'	'123,45 F'	French francs, fixed
'$###,###.##'	' $1,234.56'	',' is discretionary, '$' floats
'$###,###.##'	' $12.34'	
'mm/dd/yy'	'01/16/26'	month is padded with 0's
'MM/dd/yy'	' 1/16/26'	month is padded with spaces
'mm/dd/yyyy'	'01/16/1926'	
'MM/dd/yyyy'	' 1/16/1926'	
'dd/mm/yyyy'	'16-01-1926'	British format
'hh:mm'	'23:30'	time in military format
'hh:mmt'	'11:30p'	time in am/pm format
'HH:mm te'	' 1:30 pm'	hour is padded with spaces

You may have noticed in the examples given here that there is no mixing of picture characters within subfields. For instance, there are no examples of masks like "!XXXX" (convert first character to upper case, leave others alone). Although picture masks like these are legal, they are not guaranteed to produce the desired results.

A typical application of a user-defined character set might be to limit entry in a character field to a narrow range of non-contiguous values. For example:

```
UserSet1 := ['D', 'R', 'I']; {Democrat, Republican, Independent}
ForceCaseUser['1'] := UpperCase;

UserSet2 := ['m', 'f'];       {Male or Female}
ForceCaseUser['2'] := LowerCase;

UserSet3 := ['M', 'm', 'F', 'f'];
ForceCaseUser['3'] := NoChange;
```

Given these statements, a picture mask of '1' would allow only 'D', 'R', and 'I' to be entered into the field and input would be forced to upper case, while a picture mask of '2' would allow only 'm' and 'f' and input would be forced to lower case. The example for '3' is similar to that for '2', except that the case of input characters would not be changed. Although there are other ways you could validate entries into such fields, this is usually the easiest.

International Currencies and Conventions

As you may have noticed in some of the picture mask examples above, TPENTRY goes to some extraordinary lengths in order to handle as many world currencies and

date/time conventions as possible. In order to take advantage of this feature, there are only a few things you need to know.

First, there are seven formatting characters involved here: ':' (time separator), '/' (date separator), '.' (decimal point), ',' (thousands separator), 'c' (fixed currency symbol--left side), 'C' (fixed currency symbol--right side), and '$' (floating currency symbol). Second, these correspond to six global typed constants: ColonChar (':'), SlashChar ('/'), DecimalChar ('.'), CommaChar (','), CurrencyLtStr ('c' and '$'), and CurrencyRtStr ('C').

The third thing you need to know is that TPDATE and TPENTRY provide routines that make it easy for your program to adjust to different countries' conventions: InternationalDate and InternationalTime in TPDATE, and InternationalCurrency in TPENTRY. Using DOS's Get Country Information service, these routines are able not only to initialize the four typed constants properly for you, but also to construct a picture mask for you that is optimized for the current situation.

More complete descriptions of these routines are provided later in the chapter, and for some people they should prove indispensable. But since most programs are written to be used in a single country, we will spend just a few more moments explaining how to construct proper picture masks manually. Because date and time picture masks are discussed thoroughly in the documentation for TPDATE, we'll concentrate on issues related to currency.

Let's say that you need to construct a picture mask for a field that will contain a value in the range 0.00 to 999,999.99. Here are a few examples of how you would configure TPENTRY to deal with a given country's currency:

USA

```
CurrencyLtStr    = '$';              {default}
CurrencyRtStr    = '';               {default}
DecimalChar      = '.';              {default}
CommaChar        = ',';              {default}
PictureMask1     = '$999,999.99';    {floating currency symbol}
Result           = '  $1,234.56'
PictureMask2     = 'c999,999.99';    {fixed currency symbol}
Result           = '$ 12,345.67'
```

```
Germany
-----------------------------------------------------------------------
CurrencyLtStr      = 'DM';
CurrencyRtStr      = '';
DecimalChar        = ',';
CommaChar          = '.';
PictureMask1       = '$$999,999.99';
Result             = '  DM1.234,56'
PictureMask2       = 'cc999,999.99';
Result             = 'DM 12.345,67'

France
-----------------------------------------------------------------------
CurrencyLtStr      = '';
CurrencyRtStr      = ' F';
DecimalChar        = ',';
CommaChar          = ' ';
PictureMask2       = '999,999.99CC';
Result             = ' 12 345,67 F'
```

As you can see, TPENTRY can adapt to currency symbols of varying length (1-5 characters) on either side of a number. The key to getting the proper output is to insure that the picture mask is coordinated with the values of CurrencyLtStr and CurrencyRtStr. If the currency symbol is a single character on the left, then CurrencyLtStr should contain that character and a single 'c' or '$' should appear at the beginning of the picture mask. If it is a two-character string on the right, then CurrencyRtStr should contain those two characters and 'CC' should appear at the end of the picture mask.

Fields and Subfields

Before leaving the subject of picture masks, we need to define two key terms. "Field" is used to refer to a section of the edit screen whose characteristics were defined by a call to a routine such as AddStringField. A field can have its own prompt, its own colors, its own picture mask, and so on.

A "subfield" is either an entire field or a region in a field that is bounded (according to the picture mask) either by a literal character or one of the edges of the field. For instance, in our example program the field corresponding to MyRec.Phone had a picture mask that looked like this: '(999) 999-9999'. Only the slots containing '9' could be changed during editing. Thus, this field consisted of three subfields: the first bounded by '(' and ')', the second by ' ' and '-', and the third by '-' on the left and the end of the field on the right.

Note, however, that these generalizations do not apply when you are using the numeric and multiple-choice editors (described below), which do not discriminate between subfields and fields.

The Three Field Editors

TPENTRY has three "field editors" for handling keyboard input. The main one is the string editor, which can be used for almost any data type. The numeric editor is a calculator-style editor that allows numbers to be entered from right to left. And the multiple-choice editor handles cases in which the user is expected to choose from a short list of acceptable choices rather than actually entering text. A complete list of the commands available in each of the three editors appears in the next section. The following provides a brief overview of the three editors.

The string editor is very much like the line editor in TPEDIT. The major differences are that the string editor in TPENTRY has a larger command set, in order to allow movement from field to field, and it must deal with picture masks. Although it works with a wide variety of data types, including all the standard numeric types, it is oriented toward the editing of strings rather than numbers. It is particularly limited in its ability to handle picture masks containing discretionary commas and floating dollar signs. When editing fields whose picture masks contain either of these formatting characters, the string editor must temporarily strip out these characters while editing, then put them back after a number has been entered.

The numeric editor is designed exclusively for entering and editing numbers. It works just like a calculator, meaning that the cursor is always positioned at the far right edge of the field and digits are always inserted at the end, pushing previously entered digits to the left. Unlike the string editor, it correctly displays discretionary commas and floating dollar signs as text is being entered, rather than inserting them after the fact. It is especially well-suited for applications in which long series of numbers must be entered one after the other.

The multiple-choice editor is not an editor in the normal sense of the term, since it does not allow text to be entered or altered. Rather, it allows users to choose from a short list of alternatives by pressing a particular key until the desired choice appears on screen. It can also be used with numeric fields in cases where you want the user to be able to increment or decrement a value in predetermined stages, rather than actually entering a value. An added advantage of using the multiple-choice editor in these cases is that it is possible (as ENTRY.PAS demonstrates) to update the values of related fields instantly each time the user increments or decrements the value.

Editing Commands

This section briefly describes all of the commands recognized by the three field editors. These commands fall into three categories: editing commands, field movement commands, and exit commands. In the sections that follow, the symbolic name for each command (if any) is shown, then the key(s) it is assigned to, then the description of the command. The abbreviations *SE* (String Editor), *NE* (Numeric Editor), and *ME* (Multiple-choice Editor) are used when pointing out differences in

behavior among the three editors. If no abbreviations appear in a description, the command does the same thing in all three editors.

Note that the commands that move the cursor from field to field will always skip over or ignore protected fields (explained in a later section). For example, the command to move the cursor to the first field in the screen will actually move it to the second if the first field is protected. Somewhat similarly, commands that move the cursor from field to field can be affected by the "wrap mode" for the edit screen. See the discussion of SetWrapMode for details.

ESleft **<Left>, <CtrlS>**
SE: Cursor left one character. If the cursor is at the beginning of a subfield, it is moved to the end of the previous subfield.
NE, *ME*: Move cursor to previous field.

ESright **<Right>, <CtrlD>**
SE: Cursor right one character. If the cursor is at the end of a subfield, it is moved to the beginning of the next subfield.
NE, *ME*: Move cursor to next field.

ESwordLeft **<CtrlLeft>, <CtrlA>**
SE: Cursor left one word. If the cursor is at the beginning of the first word in a field, it is moved to the beginning of the last word in the previous field. A "word" is any series of non-blank characters followed by one or more blanks.
NE, *ME*: Move cursor to last word in previous field.

ESwordRight **<CtrlRight>, <CtrlF>**
SE: Cursor right one word. If the cursor at the end of a field, it is moved to the beginning of the first word in the next field.
NE, *ME*: Move cursor to first word in next field.

EShome **<Home>, <CtrlQ><S>**
SE: Cursor to beginning of field.

ESend **<End>, <CtrlQ><D>**
SE: Cursor to end of field.

ESdel **<Del>, <CtrlG>**
SE, *NE*: Delete character at cursor.

ESback **<Bksp>, <CtrlH>, <CtrlBksp>**
SE: Delete character to left of cursor.
NE: Delete character at cursor.

ESdelLine **<CtrlY>**
SE, *NE*: Clear entire field.

ESdelEol **<CtrlEnd>, <CtrlQ><Y>**
SE: Delete from cursor to end of subfield.

ESdelBol **<CtrlHome>**
SE: Delete from beginning of subfield to cursor.

ESdelWord **<CtrlT>**
SE: Delete word to right of cursor.

ESins **<Ins>**
SE, *NE*: Toggle insert mode on and off. Fat cursor indicates insert mode; thin cursor indicates overtype mode.

ESctrlChar **<CtrlP>**
SE: Insert control character. For example, to insert a ^G, you would enter <CtrlP><CtrlG>. This command is available only if the picture mask allows it, and if the video attribute for control characters is different than that for normal characters. If EntryMapCtrls is True (the default setting), control characters are displayed as uppercase alphabetics (^G is displayed as 'G', for example) in the specified attribute. If EntryMapCtrls is False, the characters are displayed normally (^O as a snowflake, for example) in the specified attribute.

ESrestore **<CtrlR>, <CtrlQ><L>**
Restore original contents of the field and continue editing.

EShelp **<F1>**
Help. If a user-written help routine was established by setting EntryHelpPtr to a value other than nil, pressing <F1> will call that routine with the help index for the current field. If EntryHelpPtr is nil, this command does nothing.

ESnextField **<Enter>, <CtrlM>**
Accept contents of field and move to the next one.

ESupField **<Up>, <CtrlE>**
Move cursor to last field above the current one that starts at a screen column less than or equal to the column where the cursor is currently located.

ESdownField **<Down>, <CtrlX>**
Move cursor to last field below the current one that starts at a screen column less than or equal to the column where the cursor is currently located.

EStab **<Tab>, <CtrlI>**
Move cursor to the beginning of the next subfield.

ESbackTab **<ShiftTab>**
Move cursor to the beginning of the previous subfield.

ESfirstFld **<CtrlPgUp>, <CtrlQ><R>**
Move cursor to first field on screen.

ESlastFld **<CtrlPgDn>, <CtrlQ><C>**
Move cursor to last field on screen.

ESpageUp
If the entry screen scrolls vertically, ESpageUp moves the cursor to the top of the previous page. If the entry screen doesn't scroll vertically (either because there aren't enough rows, or because TpEntryScrolls is not defined), ESpageUp moves the cursor to the first row on the entry screen. Note that this command is not assigned to a particular key by default. If you want to use it, you must assign it to a key--presumably <PgUp>--using AddEntryCommand, like so:

```
if not AddEntryCommand(ESpageUp, 1, $4900, 0) then ;
```

Keep in mind, however, that doing so disables the ESprevRec command--normally assigned <PgUp>--so you will probably want to reassign this command to a new key at the same time.

ESpageDown
If the entry screen scrolls vertically, ESpageDown moves the cursor to the top of the next page. If the entry screen doesn't scroll vertically (either because there aren't enough rows, or because TpEntryScrolls is not defined), ESpageDown moves the cursor to the last row on the entry screen. Note that this command is not assigned to a particular key by default. If you want to use it, you must assign it to a key--presumably <PgDn>--using AddEntryCommand, like so:

```
if not AddEntryCommand(ESpageDown, 1, $5100, 0) then ;
```

Keep in mind, however, that doing so disables the ESnextRec command--normally assigned <PgDn>--so you will probably want to reassign this command to a new key at the same time.

ESdone **<CtrlEnter>, <CtrlK><D>, <CtrlK><Q>**
Quit editing. Indicates that the user wants to end the editing session.

ESprevRec **<PgUp>**
Indicates that the user wants to quit editing the current record and start editing the previous one. Meaningful only in database-type applications where a single entry screen is used for editing multiple records.

ESnextRec **<PgDn>**
Indicates that the user wants to quit editing the current record and start editing the next one.

ESquit **<Esc>, <CtrlBreak>**
Abort. Indicates that the user wants to cancel all edits made to the current record. The original value of the current field is automatically restored. It is up to the programmer to restore the original values of the other fields.

ESincChoice **<+>, <Space>**
ME: Select next choice, or increment value.

ESdecChoice **<->**
ME: Select previous choice, or decrement value.

N.A. **<.>**
SE: If the field being edited contains a fixed decimal point, entering a period causes the cursor to be moved to the space immediately after the decimal point. Note that this command is actually assigned not to <.> but rather to the current value of DecimalChar, so if DecimalChar = ',' (as it would in some countries), then the command would be assigned to <,>.

N.A. **<->**
NE: Change the sign of the number being entered.

Note that you can alter any of these key assignments that you wish (except the last two). See the section on "Changing Key Assignments" and Appendix B for details.

Determining the Last User Command
The global variable LastEntryCommand can be used to determine, from within a validation routine, what command was last issued by the user. It can also be used for other, more sinister purposes from within a post-edit routine. If you need to force the cursor to jump to another field, you can do so like this:

```
{cancel the user's last command}
LastEntryCommand := ESnone;

{jump to new field}
ESR.CurrentField := FindFieldID(ESR, TargetFieldID);
```

where ESR is the first parameter passed to your post-edit routine, and TargetFieldID is the ID for the field to jump to. Similarly, you can force an exit from the screen editor by setting LastEntryCommand to any exit command (ESquit or ESdone, for example). Neither of these dirty tricks is encouraged, but you may use them when no cleaner alternative presents itself.

LastEntryCommand can also be changed from within a pre-edit routine in order to force a field editor to execute a particular command. This allows you, for example, to display a pick list from within your pre-edit routine, move the user's choice into the variable associated with the current field, then use the statement:

```
{force the field editor to exit immediately; move cursor to next field}
LastEntryCommand := ESnextField;
```

to force the field editor to exit immediately and move the cursor to the next field. It is also possible (and safe) to change the CurrentField from within a pre-edit routine (forcing the cursor to jump to another field):

```
{jump to new field}
ESR.CurrentField := FindFieldID(ESR, TargetFieldID);
```

But, if you do so, any command assigned to LastEntryCommand is ignored because the field editor is not called for the field that was current when your pre-edit routine was called.

Required and Protected Fields

Normally, any field in a data entry screen may be left empty. In some cases, however, empty fields are simply unacceptable, so in these situations you must designate the field as required. To do so, you just call SetRequired before and after adding such a field:

```
SetRequired(On);
{add one or more required fields}
SetRequired(Off);
```

When a field is marked as required, the data screen manager checks its contents each time it is edited, as well as when the user exits from the entry screen. If all of the non-literal characters in the field are blanks, the screen manager generates an error. Be forewarned that no error will be reported if the field is partially full. If you need to be certain that a field contains no blank spaces, you will need to specify a special validation routine such as ValidateNoBlanks when you add the field.

Similarly, there may be cases where you want a particular field to be protected from changes. That is, you want its prompt and current value to be displayed, but you

don't want to allow the user to change the value. In these cases, you would call SetProtection:

```
SetProtection(On);
{add one or more protected fields}
SetProtection(Off);
```

The example program ENTRY.PAS demonstrates the use of protected fields.

Calculated Fields

In some cases you may want a data entry screen to contain a protected field whose value is a function of the value of one or more of the other fields. Such fields are called calculated fields. Although it isn't possible for TPENTRY to handle calculated fields for you automatically, it does provide a facility that makes it easy to implement them.

The key is in the procedure SetPostEditPtr, which lets you specify a pointer to a procedure that will be called immediately after each edit, just before the cursor is moved to another field. By checking the CurrentID portion of the edit screen record that is passed to it, this procedure can determine what field was just edited, and recalculate the values for any fields affected by the change. In outline, such a procedure would look like this:

```
{$F+}
procedure CalculatedFieldHandler(var ESR : ESrecord);
begin
  {check current field}
  case ESR.CurrentID of
    {update any fields affected by changes to the current field}
    {update the screen using DrawField}
  end;
end;
{$F-}
...
SetPostEditPtr(ESR, @CalculatedFieldHandler);
```

After a calculated field's value has been changed, you can update the screen by calling the DrawField routine with the ID for the field that was altered:

```
DrawField(ESR, FieldID);
```

Note that CalculatedFieldHandler would also be called each time that a value in a mutiple choice field changed, to allow you to update the values of dependent fields while the user is making selections as well as after.

Multiple-Choice Fields

As with calculated fields, TPENTRY requires some assistance in handling multiple-

choice fields. When you call AddChoiceField to add such a field, it asks for a pointer to a routine that can convert values to strings and increment or decrement a value after the user asks for another choice. In outline, such a routine would look something like this:

```
{$F+}
procedure IncValue(var Value; ID : Byte; Factor : Integer; var S : string);
begin
  case Factor of
    +1 : {increment Value} ;
    -1 : {decrement Value} ;
  end;
  S := {string representation of Value} ;
end;
{$F-}
```

It is up to IncValue to do whatever typecasting is necessary to work with Value. It can either use an explicit typecast, such as 'Byte(Value)', or it can declare an absolute variable of the appropriate type--'ByteValue : Byte absolute Value'--and work with it. Factor indicates whether Value needs to be incremented or decremented before converting it to a string; if the Factor is 0, only conversion is necessary. (Factor will always be in the range -1..1.) ID indicates which field in an entry screen Value is associated with, making it easy to write a single routine that handles all multiple-choice fields in an entry screen.

Although this scheme was designed to handle enumerated types such as 'Gender = (Male, Female, Unknown)', it can also be used for editing numeric fields that you want the user to change only in fixed increments. Neither AddChoiceField nor the multiple choice editor know or care about the type of the field to be edited. The only restriction they impose is that it can be no larger than 10 bytes. For example, you might have a field of type Integer that was incremented or decremented by multiples of 100 rather than 1 (100*Factor rather than 1*Factor).

The example program ENTRY.PAS demonstrates both uses of multiple-choice fields (enumerated types and numeric types).

Nested Data Entry Screens

TPENTRY provides two means of working with data entry forms that are too big to fit on the physical screen. The first is a hook that allows you to create nested data entry screens. Here's a quick example of what we mean: Say you have a large database record that contains several fields that store information about a person's employment history. Rather than crowding these fields into the main entry screen, you might want to display a dummy field called "Employment history". When the user moves the cursor to that field and presses <Enter>, a second data entry screen, one containing all the fields relating to the subject, could be displayed in a popup

window. After the user has finished entering data into the second entry screen, you could then resume editing in the main entry screen.

Creating nested data entry screens requires three steps. The first is to add the dummy field to the main entry screen using AddNestedField. The second is to create the secondary entry screen and to write a procedure that will handle the calls to EditScreen for it. And the third is to modify the code that processes the exit commands that are returned by EditScreen when the fields in the main entry screen are being edited. If the exit command is ESnested, you would call the procedure that you wrote in Step 2. For an example of how to do this, see ENTRY.PAS.

We should also point out that the hook provided for nested entry screens could be used in other situations as well. For example, let's say that one field in a data record is a large buffer set aside for "Notes," and that you want to use Borland's Binary Editor (included with the Editor Toolbox) for editing the notes. The best way to do this is to create a dummy Notes field in the entry screen using AddNestedField, and invoke the editor when the ESnested command is returned.

The keys to success in this situation are the Buffer and EOtext fields in an Editor Control Block (EdCB). At the point where you would normally load a file into the editor, you would instead do three things: 1) copy the text in your Notes field into the editor's buffer, 2) specify the amount of text moved into the buffer, and 3) append a ^Z to the end of the text. The code might look something like this:

```
Move(MyRec.Notes, MyEdCB.Buffer^, MyRec.BytesToMove);
MyEdCB.EOtext := MyRec.BytesToMove;
{Buffer points to a *0-based* array}
MyEdCB.Buffer^[MyRec.BytesToMove] := ^Z;
```

At the point where you would normally save the file being edited, you would reverse the process:

```
MyRec.BytesToMove := MyEdCB.EOtext;
Move(MyEdCB.Buffer^, MyRec.Notes, MyRec.BytesToMove);
```

Scrollable Data Entry Screens

The second way to work with forms that are too large to fit on the screen is to specify a *data entry window* using SetEntryWindow. This procedure allows you to specify a region of the screen that is to be used for displaying the data entry screen. Let's say that the window contains 15 rows, but you have added fields with row coordinates as high as 50. The screen manager would then scroll the window as necessary any time the cursor is moved to a field that is not currently being displayed inside the window.

When using this facility, there are a few key points to bear in mind. First, although it

is generally OK for the prompt for an edit field to be on a different screen row than the field itself, this is not always the case when working with a scrollable entry screen. To avoid extensive overhead, the screen manager assumes that there are no prompts on a row prior to the first edit field, nor any lower than the last field. This assumption is unimportant if the entry screen doesn't scroll, but it should be kept in mind if it does.

Second, when adding fields to an entry screen, remember that the coordinates are always treated as relative to the data entry window. When InitESrecord is called, it sets the default coordinates for the window to be the entire screen. Thus, if SetEntryWindow is never called, the field coordinates will effectively be treated as absolute (so long as all the fields will fit on screen at once). But if you were to execute, for example, the statement

```
SetEntryWindow(MyESR, 1, 5, 80, 20, Framed, TextAttr, TextAttr);
```

then the coordinates for any fields added to MyESR would be treated as relative to Row 5, Column 1 of the physical screen.

Like the popup windows created by TPWINDOW, data entry windows may be framed or unframed. If the window is framed, and mouse support has been activated by a call to EnableEntryMouse, TPENTRY will use the right edge of the frame as a scroll bar, assuming that the data entry screen is larger than the window. Moving the mouse cursor to the top right corner of the window and clicking left will scroll the display up one row; clicking the left button on the bottom right corner will scroll the display down one row. More rapid movements from one part of the entry screen to another can be made by moving the mouse cursor to the appropriate position along the scroll bar and clicking left.

Since most applications don't require scrollable entry screens, conditional compilation directives are used to prevent the code that handles scrolling from being pulled in when it isn't needed. So, if you wish to create scrollable entry screens, you must (1) load TPDEFINE.INC into your editor, (2) delete the '.' in

```
{.$DEFINE TpEntryScrolls}
```

at the end of the file, (3) save the file, and (4) recompile TPENTRY.PAS. You can then create scrollable entry screens simply by specifying row and/or column coordinates that are outside the bounds of the entry screen's window. By default, the entire screen is the entry window; to limit it, use SetEntryWindow. TPENTRY handles everything else automatically.

To give you an idea of what a large scrollable entry screen looks like, see the demo program, BEST.PAS (Big Entry Screen Test). It doesn't do anything useful; it just

shows you a little bit of what TPENTRY can do. The special features worth noting are:

- <AltS> lets you toggle between line- and page-based scrolling (see the discussion of ScrollByPage in TPENTRY:Declarations).
- The entry in the first field determines how many rows below it are visible. This is to demonstrate a technique described below in the section "Hidden and Semi-Hidden Fields".

Before trying to compile BEST.PAS, be sure to define TpEntryScrolls in TPDEFINE.INC.

Displaying Help Messages

Just as TPENTRY lets you specify a routine to be called *after* a field has been edited (see the discussion of "Calculated Fields," above), it also allows you to specify a routine to be called immediately *before* editing takes place. Although there are many things you might want to do in such a routine, the most obvious is to display a help message explaining what the user is supposed to enter for a given field. To set up the routine, all you have to do is call SetPreEditPtr with the name of your edit screen record and the address of your routine. The routine itself should be of the following form:

```
{$F+}
procedure DisplayHelpMessage(var ESR : ESrecord);
begin
  case ESR.CurrentID of
    {display help message for this field}
  end;
end;
{$F-}
...
SetPreEditPtr(ESR, @DisplayHelpMessage);
```

For an example of a routine that displays help messages this way, see ENTRY.PAS.

Error Handling

Error messages produced by TPENTRY may be handled in a similar fashion. To install an error handler, call SetErrorPtr with the name of your edit screen record and the address of the routine that will display the error messages. Any time an error occurs, TPENTRY will pass the routine an error code and a generic message such as 'Number is out of range.' or 'Invalid number format.'. The error code indicates the type of error that occurred (see the list of constants in TPENTRY.PAS labelled "error message codes"), allowing you to substitute your own error messages if you don't like the defaults.

Your error handler should be of the following form:

```
{$F+}
procedure ErrorHandler(var ESR : ESrecord; Code : Byte; Msg : string);
begin
  {display error message and wait for keypress}
  {clear portion of screen overwritten by message}
end;
{$F-}
...
SetErrorPtr(ESR, @ErrorHandler);
```

If you have not installed any error handler and an error occurs, TPENTRY will first "ring the bell" and then either resume execution, if the error was due to an invalid entry by the user, or halt if the error was fatal (an out of memory error, for example).

For an example of a routine that displays error messages, see ENTRY.PAS.

Background Tasks
Like several other units in Turbo Professional, TPENTRY makes it easy to perform background tasks while waiting for keyboard input. See Appendix B for details, and ENTRY.PAS for an example.

Context-sensitive Help
Although context-sensitive help is not an inherent feature of TPENTRY, it does provide hooks to allow you to implement sophisticated help systems easily. The first step is to assign each field in each data entry screen its own unique number (1-10000), and pass that number as the HelpIndex parameter to the routine that adds that field to the entry screen. The remaining steps are explained in Appendix B. ENTRY.PAS demonstrates how to make TPENTRY and TPHELP work together smoothly.

Mouse Support
Like many of the screen-oriented units in Turbo Professional, TPENTRY provides built-in mouse support. During data entry, the following functions are assigned to the mouse:

ESmouse **<Click Left>**
Select new field using mouse. This command allows you to move rapidly to another field on the data entry screen. Just move the mouse cursor to the desired field, then click the left mouse button. If the field moved to has a nested data entry screen associated with it, EditScreen will exit with a return code of ESnested, invoking the nested screen. If the current field is a multiple-choice field, moving the mouse cursor to it and clicking the left button will increment

the value. If the current field is a boolean or yes/no field, moving the mouse cursor to it and clicking the left button will toggle the value. Finally, if the current field was specially flagged (by making a call to SetExitOnSecondClick), moving the mouse cursor to it and clicking left will cause EditScreen to exit with a return code of ESclickExit. (The example program ENTRY.PAS demonstrates all of these features. ENTRY.PAS is documented in Chapter 13.)

ESquit **<Click Right>**
Abort. Same as pressing <Esc>.

EShelp **<Click Both>**
Help. Same as pressing <F1>.

ESnextField **<Click Center>**
Same as pressing <Enter>. (Three-button mice only.)

Note that the functions assigned to the mouse may be changed in precisely the same manner as the key assignments. See Appendix B for details.

Now, how do you implement mouse support in a program that uses TPENTRY? We will assume that you've already read the documentation for the TPMOUSE unit (if you haven't, please do), which discusses the more general topics pertaining to mouse support, and cover only those issues that pertain especially to TPENTRY.

First and foremost, it is important to select a software mouse cursor at the beginning of your program, because TPENTRY needs the hardware cursor for its field editors. Second, you need to call EnableEntryMouse to enable mouse support. And finally, if you write your own keyboard input routine in order to implement a background task, be sure to follow the guidelines in Appendix B.

Probably the best way to learn how to use TPMOUSE and TPENTRY together is to study the source code for the demonstration program ENTRY.PAS. Like TPENTRY itself, ENTRY uses a conditional compilation directive found in TPDEFINE.INC, 'UseMouse', that determines whether mouse support is implemented or not. Using an editor or file browser, you may wish to search for all instances of that directive in order to see precisely what had to be done in order to add mouse support to the program.

User-defined Types
Although TPENTRY provides built-in support for an exceptionally wide range of data types, there may be times when you want to use it with a data type that it doesn't directly support. Fortunately, TPENTRY provides the means for you to do this. The process involves three basic steps.

First, you have to write a conversion routine to translate your data type into a string that can be displayed and edited. Second, you have to write a validation routine that can determine whether or not a particular entry was valid. (Both of these topics are discussed below.) And finally, you need to call AddUserField to add any fields of this new type to your edit screen.

Conversion Routines

A conversion routine is a procedure that translates a variable of a particular type into a string and vice versa. TPENTRY has conversion routines for all the data types that it supports, but if you need to use TPENTRY with a user-defined type, you'll have to write your own conversion routine to go with it. Space does not permit a complete explanation of how to do this, so you will need to study the source code for TPENTRY in order to learn how to do it. At least one of the existing routines should serve as an appropriate model for your own. We can tell you, however, that a conversion routine must be of the following form:

```
{$F+}
procedure ConversionRoutine(var FR : FieldRec; PostEdit : Boolean);
begin
  if PostEdit then
    {convert FR.EditSt^ to FR.VarPtr^}
  else
    {convert FR.VarPtr^ to FR.EditSt^} ;
end;
{$F-}
```

The FR parameter is a record containing all relevant information about the field. The PostEdit parameter tells you whether you need to convert the variable to a string or vice versa.

Validation Routines

A validation routine is somewhat similar to a conversion routine, except that its purpose is to determine whether or not a string that was just edited represents a valid entry for a particular field. Such a routine might determine, for example, whether a number that was entered is in a particular range, whether a field is partially empty when it needs to be full, or whether a string appears in a list of valid responses.

Again, space does not allow a complete explanation of what is required to write a validation routine, so you will need to study the examples in the source code. Two special-purpose validation routines, ValidateNotPartial and ValidateNoBlanks, should be of particular interest. In any case, a validation routine should be of the following form:

```
{$F+}
function ValidityCheck(var FR : FieldRec;
                       var ErrCode : Byte;
                       var ErrorSt : StringPtr) : Boolean;
begin
  {strip extraneous picture characters out of FR.EditSt^, if necessary}
  {convert the string to the appropriate type, if necessary}
  {verify that the result is a valid response}
end;
{$F-}
```

The actual steps involved, however, will depend on the data type.

There is one fairly common case that requires some explanation, the case where an entry is valid only if it appears in a list of acceptable answers. ENTRY.PAS includes an example of how to write this kind of validation routine. If you look at the ValidState routine, you'll see that (among other things) it scans a list of valid state abbreviations checking to see if the user's entry appears in the list. It uses the global variable LastEntryCommand to determine what the last command was. If it was ESuser1 (<F2>) or ESclickExit (<ClickLeft>), no validation is performed because those commands indicate that the pick list of state abbreviations is about to be displayed. Note that the call to ValidateNotPartial within the routine is really unnecessary in this case. It is included primarily to demonstrate how multiple types of validation can be performed within a single validation routine. If the user enters a single letter (e.g., 'C'), the error message will be 'Partial entries not allowed here.'. If the field is full but the abbreviation is invalid (e.g., 'CS'), the error message will be 'Not a valid abbreviation for a state.'.

When writing your own validation routines, don't forget that they must be compiled under the FAR model (using {$F+}) and they may not be nested within another procedure. This same rule applies, of course, to any user-written procedure that is going to be called from a routine in another unit.

Changing Key Assignments

There are several ways that you can change the keys that field editor commands are assigned to. See Appendix B for details.

It is also possible to disable particular commands without changing any key assignments. The global typed constants StringCommands, NumberCommands, and ChoiceCommands contain the set of commands that are valid for each of the three editors, so you can disable a command simply by changing the appropriate set.

Compiling TPENTRY

Before you compile TPENTRY, there are a few things you need to know. First, two categories of routines are normally disabled via conditional compilation directives:

those for working with 8087 reals (Single, Double, Extended, Comp), and those for working with BCD reals (as implemented in TPBCD). There are also two categories of routines that can easily be disabled if you want to reduce the size of the unit: those for working with multiple-choice fields, and those pertaining to the numeric (right-to-left) editor. Finally, there is a directive, defined by default, that enables or disables mouse support.

To enable or disable use any of these features, you will need to modify one of two files: TPENTRY.PAS (to add support for 8087 reals) or TPDEFINE.INC (to change any of the other directives). You can disable a particular feature by inserting a period between the left brace and the dollar sign, e.g.:

```
{.$DEFINE IncludeNumeric}
```

or enable the feature by removing the period. By default, the numeric editor and multiple-choice fields are enabled, while support for BCD and 8087 reals is disabled.

Hidden and Semi-Hidden Fields

Several customers have requested ways to temporarily hide and unhide fields, and to suppress the display of a field's value in certain situations. The ChangeFieldAttr, ChangePromptAttr, and ChangeStringAttr procedures make it relatively easy to handle these cases. Below are descriptions of some common situations and brief explanations of how to handle them. Note that all these descriptions assume that you understand "Calculated Fields" earlier in this section (where the concept of the post-edit routine is introduced).

(1) You have a calculated field of type Real used to display profit/loss. If the value is negative, you want the field to be displayed in red; if it is positive, in black. The solution here is to have your post-edit routine (which is already calculating the value of the field) call ChangeFieldAttr to set the field attribute based on the current value. If you're initializing the entry screen once and using it to edit multiple records (as is done in ENTRY.PAS), don't forget to initialize the field attribute each time you edit a new record.

(2) Same as (1), but the field is not protected, and you want the field displayed in red (if it is negative) even while editing it. In this case you would call ChangeStringAttr as well as ChangeFieldAttr.

(3) You have a numeric field whose initial value is 0, and you want the field to appear as a complete blank until its value is non-zero. The solution to this problem is demonstrated by ENTRY.PAS in its handling of the Wage field, and it is essentially identical to that described for (1) above. The main difference is in the choice of attributes. When the field's value is 0, you simply set the

foreground portion of the field attribute equal to the background color: if the background color is black, you would use $00; if it is blue, you would use $11; if it is light gray, you would use $77; and so on. When the field's value is non-zero, change the field attribute to something that will make the contents of the field visible. In either case, you would then redraw the field with DrawEditField. (See the SetWageFieldAttribute and UpdateHandler procedures in ENTRY.PAS.)

(4) You have a calculated field that you want to hide completely when its value is 0. The solution is essentially the same as the one for (3), except that you need to hide the prompt (using ChangePromptAttr) as well as the field itself. ENTRY.PAS demonstrates the technique in its handling of the Age field. (See the SetAgeFieldAttribute and UpdateHandler procedures in ENTRY.PAS.)

(5) You have a field that is needed only if the value in one of the other fields is within a certain range. When it is not needed, you don't want it displayed at all. The solution here is similar to the one for (4), except that you also need to use ChangeProtection to protect the field when it is hidden and unprotect it when it is unhidden.

(6) Your entry screen has a variable number of fields, with the actual number of fields depending on a value entered by the user in one of the fields in the same entry screen. Although this sounds like a complex case, it is essentially the same as case (5) except that there are multiple fields to be hidden/unhidden at a time rather than one, and consequently you need to use ChangeProtectionFast and ResetEntryScreenFlags rather than ChangeProtection. For an example of how to handle this situation, see the UpdateHandler and HideFields routines in BEST.PAS.

A Few Words of Warning

TPENTRY can get bogged down if the entry screen has a large number of fields, partly due to the flexibility inherent in TPENTRY's design (the flexibility that allows for user-definable data types, validation/conversion routines, picture masks, etc.). On a '286 machine, performance is generally bearable with 400 fields, as BEST.PAS demonstrates, but you'll still see some brief delays when the entire entry screen has to be redrawn. On slower machines, the threshold of pain will be somewhat lower. It is therefore recommended that you break large entry screens up into multiple, nested entry screens if at all possible.

You should also keep in mind that, although TPENTRY allows up to 255 rows and up to 255 columns, you cannot create an 255 x 255 entry screen. Those limits are really somewhat arbitrary (as is the limit of 2000 fields per entry screen). The actual limit is imposed by Turbo Pascal's 64K limit on individual data structures. Since the virtual screen used to store an image of the entire entry screen cannot exceed that

limit, the product of the highest row and column values for the entry screen cannot exceed 32,760 ($FFF1 div 2). In all likelihood, though, performance will become a major issue long before you reach any of these limits.

Horizontal scrolling of the entry screen, if it is necessary at all, takes place only when you move from one field to another. It never occurs while editing a particular field. Although TPENTRY tries to insure that the full edit field can be displayed within the edit window, it does not generate an error if the edit field is wider than the window. It is up to you to ensure that this is not the case. Keep in mind, however, that the string editor in TPENTRY can do horizontal scrolling of a different sort when you need to edit strings longer than the width of the window. See the discussion of the EditLen parameter to AddStringField.

Undocumented Routines

The following low-level routines are interfaced by TPENTRY but are not documented. They are interfaced primarily for the convenience of advanced programmers who need to write special validation and conversion routines to handle unsupported data types. If you understand TPENTRY well enough to know that you need one of these routines, we assume that you can learn how to use them by studying the source code for the unit.

```
procedure AddFieldPrim(var ESR : ESrecord; var FR : FieldRec;
                       var PictureSt : string; var PromptSt : string);
  {-Low level routine to allocate and finish initializing a FieldRec}

function FindFieldID(var ESR : ESrecord; FieldID : Byte) : FieldRecPtr;
  {-Return a pointer to the FieldRec with the specified ID, or nil if not found}

procedure InitPictureFlags(var Picture : string; var PFlags : PictureFlags);
  {-Initialize a picture flags table}

procedure MergePicture(Picture, St : string; var S : string; EditFlags : Word);
  {-Merge St with Picture and return result in S}

procedure StripPicture(var Picture, EditSt, S : string);
  {-Strip picture characters out of EditSt and return in S}

{$IFDEF TpEntryScrolls}
procedure CopyVirtualScreenToWindow(var ESR : ESrecord);
  {-Copy the virtual screen to the edit window}
```

CopyVirtualScreenToWindow, intended primarily for internal use, is interfaced for use by programs that need to write to the virtual screen associated with a particular ESrecord (entry screen record), then update the physical screen. If you really need to do this, you should study the source code in TPENTRY.PAS. We will tell you, however, that you need to do something like this:

```
{switch to the virtual screen}
ActivateVScreen(ESR.VS);
```

```
{** write to the virtual screen--see ActivateVScreen in section 2.C **}

{switch back to the physical screen}
DeactivateVScreen;

{update the physical screen}
CopyVirtualScreenToWindow(ESR);
```

Please keep in mind that (1) CopyVirtualScreenToWindow does not check to be sure that a virtual screen is actually in use and (2) the virtual screen is always completely cleared and redrawn each time a call to either DrawEditScreen or EditScreen is made. Incidentally, it is not necessary to hide the mouse prior to calling CopyVirtualScreenToWindow or to unhide it afterward--that is done automatically if UseMouse is defined.

More importantly, the virtual screen is not normally created until one of those two procedures is called (EditScreen calls DrawEditScreen). It is *not* created by InitESrecord, as you might expect, because the bounds of the window are not known until all the fields are added. Instead, it is created the first time that it is needed by DrawEditScreen. If there is not enough memory to allocate it, TPENTRY simply aborts the program with an out-of-memory error. If this behavior is unacceptable, you should call the following procedure *after* adding the last field to your entry screen, *after* any calls to SetEntryWindow, and *before* the first call to either DrawEditScreen or EditScreen:

```
{$IFDEF TpEntryScrolls}
function EntryScrollCheck(var ESR : ESrecord) : Boolean;
  {-Determine if window is scrollable. If so, allocate and
    initialize virtual screen. Returns False only in case of
    insufficient memory}
{$ENDIF}
```

This function determines whether or not an entry screen is scrollable and, if it is, allocates a virtual screen for it. It returns False only if the entry screen is scrollable *and* there is insufficient memory to allocate the virtual screen for it. It returns True if a virtual screen was allocated successfully, or if none is needed. In a well-designed program that includes scrollable entry screens, you will probably want to call this function yourself so that you can handle the error more gracefully, especially if your program uses lots of heap space.

Types

```
CaseChange = (NoChange, UpperCase, LowerCase);
```

Variables of this type determine how user-defined picture mask characters are handled. See the section on "Picture Masks" at the beginning of this chapter for more details.

```
CharSet = set of Char;
```

Variables of this type typically contain a set of characters corresponding to a particular character in a picture mask, and are used for validating keyboard input.

```
CommandSet = set of EStype;
```

A variable of this type contains the set of commands that are implemented by one of the three field editors.

```
ESrecord =
  record
    ...
    CurrentID : Word; {ID of current field}
    ...
  end;
```

Variables of this type are used to store general information about a particular edit screen: the locations in memory of the various fields, the number of fields, and so on. The only field in the data structure that you are likely to use yourself is CurrentID, which contains the ID of the field that the edit screen manager is currently working with. The various uses of this information are explained at appropriate places elsewhere in this chapter.

```
EStype = ESnone..ESmouse;
```

A variable of this type is generally used to store the result of a call to the EditScreen procedure, which always returns a byte value in the range ESnone..ESquit. (See the Constants section, below.)

```
FieldRec =
  record
    FieldID   : Word;       {field ID--assigned automatically}
    ...
    Picture   : StringPtr; {points to edit mask}
    ...
    EditSt    : StringPtr; {points to string to edit}
    ...
    RangeLo   : RangeType; {valid range for the field}
    RangeHi   : RangeType;
    ...
    VarPtr    : Pointer;   {points to the actual variable}
    ...
  end;
FieldRecPtr  = ^FieldRec;
```

Variables of type FieldRec contain all pertinent information about a particular field. Most programmers will never need to know anything about these data structures. The most likely exceptions are advanced programmers who need to write special conversion and validation routines to handle unsupported data types,

and even then they will need to understand only a few of the fields in these data structures, probably the ones shown above. The FieldID is the unique number assigned to a field when it is added to an edit screen. Picture points to the picture mask for the field. EditSt points to the string that is used when editing the field. RangeLo and RangeHi contain the upper and lower bounds for the field, if any. And VarPtr points to the variable actually being edited.

```
WrapModeType = (StopAtEdges, WrapAtEdges, ExitAtEdges);
```

See the discussion of the SetWrapMode procedure later in this chapter for a discussion of the three values that make up this enumerated type.

Constants

```
ChoiceCommands : CommandSet =
  [ESchar, ESrestore, ESleft..ESwordRight, EShelp..ESmouse];
```

This variable contains the set of editing commands that are supported by the multiple choice editor. It is interfaced only to allow you to disable a particular command by removing it from the set. You should never add to this set unless you are reenabling a previously disabled command.

```
CommaChar : Char = ',';
```

This variable contains the character that is to be displayed when a discretionary comma (',') is needed in a numeric field. See the section on "Picture Masks" at the beginning of this chapter for more details.

```
CurrencyLtStr : string[5] = '$';
CurrencyRtStr : string[5] = '';
```

These strings contain the currency symbol for a given country. See the section on "International Currencies and Conventions" for details.

```
DecimalChar : Char = '.';
```

This variable contains the character that is to be displayed when a decimal point is needed in a numeric field. See the section on "Picture Masks" at the beginning of this chapter for more details.

```
EntryHelpPtr : Pointer = nil;
```

If not nil, this variable should point to a user-written routine that handles requests for help. See the section on "Context-sensitive Help" at the beginning of this chapter for more details concerning its use.

```
EntryMapCtrls : Boolean = True;
```

If True, any control characters in a field are displayed as uppercase alphabetics (^G as 'G', for example). If False, control characters are displayed normally.

```
ESnone      = 00;  {not a command}
...
ESnextRec   = 26;  {next record}
ESprevRec   = 27;  {previous record}
...
ESpageUp    = 30;  {previous page}
ESpageDown  = 31;  {next page}
ESnested    = 32;  {exit for nested form}
ESuser0     = 33;  {user-defined exit keys}
...
ESuser9     = 40;
ESdone      = 41;  {completely finished with editing}
ESquit      = 42;  {restore default string and quit}
ESclickExit = 43;  {double mouse click produces an exit}
ESmouse     = 44;  {select field pointed to by mouse}
```

Each of these constants corresponds to an editing, cursor movement, or exit command. Of the ones listed here, all but ESnone, ESmouse, ESpageUp, and ESpageDown fall into the category of exit commands, and these are probably the only ones you need to know about. ESuser0..ESuser9 are simply hooks that allow the programmer to define new exit commands. The ESnested command is returned when the current field is marked as being linked to a nested data entry screen, and the user has executed the ESenter command (normally assigned to <Enter>). The ESclickExit command is explained in the documentation for SetExitOnSecondClick. The example program ENTRY.PAS demonstrates how to use them. See the section on "Editing Commands" at the beginning of this chapter for descriptions of all other available commands.

```
ForceCaseUser : array[User1..User8] of CaseChange = (
  NoChange, NoChange, NoChange, NoChange,
  NoChange, NoChange, NoChange, NoChange);
```

The values in this array correspond to the special picture mask characters User1..User8. See the section on "Picture Masks" at the beginning of this chapter for more details concerning the use of this array.

```
InitError     = 0;   {uninitialized ESrecord}
OverflowError = 1;   {too many fields}
ParamError    = 2;   {a bad parameter to a procedure}
MemoryError   = 3;   {not enough memory}
FormatError   = 4;   {bad format}
RangeError    = 5;   {entry out of range}
ReqFldError   = 6;   {field is required}
BlanksError   = 7;   {field contains blanks}
PartialError  = 8;   {field contains partial entry}
BadCharError  = 9;   {field contains illegal character}
```

These are the error codes that could be passed to a user-defined error handler established with SetErrorPtr. Note that errors 0..3 are considered fatal, and will cause the program to abort as soon as the error handler (if any) has been called.

```
InsertMode : Boolean = True;
```

This variable indicates whether the field editors are in insert or overtype mode. It is interfaced primarily to allow a user-written keyboard handler (pointed to by EntryKeyPtr) to display the current mode.

```
NumberCommands : CommandSet =
  [ESchar, ESrestore..ESdelLine, ESins..ESmouse];
```

This variable contains the set of editing commands that are supported by the numeric editor. It is interfaced only to allow you to disable a particular command by removing it from the set. You should never add to this set unless you are reenabling a previously disabled command.

```
Off = False;
On  = True;
```

These two constants are intended to help you write self-documenting code with TPENTRY. They are typically used as parameters to procedures that turn certain editing options on or off.

```
ReadOnlyCommands : CommandSet =
  [EShome..ESwordRight, ESins..ESbackTab, ESnextField..ESmouse];
```

This variable contains the set of editing commands that are available when in read-only mode (see the documentation for the EditScreen procedure). This command set is also used when the current field is marked as a link to a nested data entry screen (see the documentation for AddNestedField). It consists of all non-destructive commands (those that do not alter the contents of a field).

```
StringCommands : CommandSet =
  [ESchar..ESmouse];
```

This variable contains the set of editing commands that are supported by the string editor. It is interfaced only to allow you to disable a particular command by removing it from the set. You should never add to this set unless you are reenabling a previously disabled command.

```
User1 = '1';
User2 = '2';
User3 = '3';
User4 = '4';
User5 = '5';
User6 = '6';
User7 = '7';
User8 = '8';
```

These special characters can be used in picture masks to restrict input to a user-defined set of characters (UserSet1..UserSet8). See the section on "Picture Masks" at the beginning of this chapter for more details concerning their use.

```
UserSet1  : CharSet = [#0..#255];
UserSet2  : CharSet = [#0..#255];
UserSet3  : CharSet = [#0..#255];
UserSet4  : CharSet = [#0..#255];
UserSet5  : CharSet = [#0..#255];
UserSet6  : CharSet = [#0..#255];
UserSet7  : CharSet = [#0..#255];
UserSet8  : CharSet = [#0..#255];
```

These character sets correspond to the special picture mask characters User1..User8. See the section on "Picture Masks" at the beginning of this chapter for more details concerning their use.

```
Subst1      = #1; {for user-defined substitutions}
Subst2      = #2;
Subst3      = #3;
Subst4      = #4;
Subst5      = #5;
Subst6      = #6;
Subst7      = #7;
Subst8      = #8;
SubstChars : array[Subst1..Subst8] of Char = (
  #1, #2, #3, #4, #5, #6, #7, #8);
```

The SubstChars array allows you to alter the characters substituted for the literal picture mask characters Subst1..Subst8 (#1..#8). For example, you could use the following technique to insert a literal '.' (normally interpreted as a decimal point) into a picture mask:

```
PictureMask := '!!!!!!!!'#1'!!!';
SubstChars[#1] := '.';
```

The resulting picture mask allows entries like 'FILENAME.EXT'.

```
{$IFDEF TpEntryScrolls}
const
  ScrollByPage : Boolean = False;
```

This typed constant allows you to select an alternate mode of scrolling. If ScrollByPage is True and the cursor is moved to a field above or below the edges of the entry window, the entry screen is scrolled by one full page (or more) rather than by a single line.

So, if the window is 20 rows high, the cursor is on the 20th row, and you press <Down> to get to a field on row 21, rows 21-40 would scroll into view. If ScrollByPage is False, as it is by default, rows 2-21 would be scrolled into view instead. Besides limiting the amount of scrolling that occurs, this option also facilitates the creation of entry screens in which related fields are grouped together in distinct "pages".

```
LastEntryCommand : EStype = ESnone;
```

Used to store the last command issued by the user.

Variables

```
EntryKeyPtr : Pointer;
```

This variable points to a user-written routine that handles keyboard input. See the section on "Background Tasks" at the beginning of this chapter for more details concerning its use.

AddArrayField

Declaration

```
procedure AddArrayField(var ESR : ESrecord;
                        Prompt : string;
                        pRow, pCol : Byte;
                        Picture : string;
                        eRow, eCol,
                        EditLen : Byte;
                        HelpIndex : Word;
                        Validation : Pointer;
                        var EditVar);
```

Purpose

Add a field of type Array (of Char) to a data entry screen.

Comments

This procedure is provided mainly for the convenience of database programmers, who frequently use an "array[1..n] of Char" in situations where other Pascal programmers might use a "string[n]".

The parameters for AddArrayField are identical to those for AddStringField except for the last one. EditVar is assumed to be an array[1..n] of Char, where n is equal to EditLen or Length(Picture), whichever is greater. See the section on AddStringField for a discussion of the other parameters.

Note that the blank trimming option is automatically turned off for fields of this type to insure that arrays are padded with blanks to their full width.

Example

```
var MyArray : array[1..8] of Char;
...
AddArrayField(ESR, 'MyArray:', 1, 1, '', 1, 10, 8, 0, nil, MyArray);
```

Add an array of char field to be displayed at the top left corner of the screen.

See Also

AddStringField

Declaration

```
procedure AddBcdField(var ESR : ESrecord;
                      Prompt : string;
                      pRow, pCol : Byte;
                      Picture : string;
                      eRow, eCol : Byte;
                      HelpIndex : Word;
                      BcdLo, BcdHi : BCD;
                      Places : Byte;
                      var EditBcd : BCD);
```

Purpose

Add a field of type BCD to a data entry screen.

Comments

This routine adds a BCD field to the specified entry screen. Its parameters are essentially identical to those for fields of other floating point types.

ESR is the edit screen to add the BCD field to. Prompt is the message to be displayed to the left (usually) of the edit field. pRow and pCol are the screen coordinates for the Prompt. Picture is the picture mask for the field. Note that this parameter is required for BCD fields. eRow and eCol are the screen coordinates for the edit field. HelpIndex is the index to be passed to the user-defined help routine, if there is one, when <F1> is pressed. BcdLo and BcdHi represent the minimum and maximum values for the BCD being edited. If they are equal, no special range checking is performed. Places is the maximum number of decimal places to be displayed. This value is ignored if the picture mask contains a fixed decimal point ('.'). EditBcd is the BCD variable to be edited.

Note: In order to use AddBcdField, you must activate the conditional compilation directive UseBCD in TPDEFINE.INC and recompile TPENTRY.

Examples

```
var Salary : BCD;
...
AddBcdField(ESR, 'Salary', 1, 1, '$999,999.99', 1, 8, 1, ZeroBcd, ZeroBcd, 0,
  Salary);
```

Salary is a field of type BCD to be displayed at the top left corner of the screen. No special range checking will be performed, although there is an implied maximum value of 999,999.99 and an implied minimum of 0 because of the width and contents of the picture mask. The Places parameter is irrelevant because of the fixed decimal place in the picture mask.

```
var Earnings : BCD;
...
AddBcdField(ESR, 'Earnings', 1, 1, '###########', 1, 10, 1, ZeroBcd, ZeroBcd,
  2, Earnings);
```

Again, no special range checking will be performed on this field, but there is an implied maximum value of 99,999,999,999 and an implied minimum of -9,999,999,999 because of the width and contents of the picture mask. The Places parameter is important because of the absence of a fixed decimal place in the picture mask.

AddBooleanField

Declaration

```
procedure AddBooleanField(var ESR : ESrecord;
                          Prompt : string;
                          pRow, pCol : Byte;
                          Picture : string;
                          eRow, eCol : Byte;
                          HelpIndex : Word;
                          var EditBool : Boolean);
```

Purpose

Add a field of type Boolean to a data entry screen.

Comments

This routine adds a Boolean field to the specified entry screen. Its parameters are essentially identical to those for routines that add numeric fields, except that it requires no parameters to indicate minimum and maximum values, and its picture mask parameter is optional. (If none is specified, a picture mask of 'B' is assumed.)

ESR is the edit screen to add the Boolean field to. Prompt is the message to be displayed to the left (usually) of the edit field. pRow and pCol are the screen coordinates for the Prompt. Picture is the picture mask for the field. Note that this parameter is optional for Boolean fields. If specified, it should consist only of literal characters (optional) and the letter 'B'. eRow and eCol are the screen coordinates for the edit field. HelpIndex is the index to be passed to the user-defined help routine, if there is one, when <F1> is pressed. EditBool is the Boolean variable to be edited.

Example

```
var Married : Boolean;
...
AddBooleanField(ESR, 'Married', 1, 1, '', 1, 9, 1, Married);
```

Married is a field of type Boolean to be displayed at the top left corner of the screen. Notice that Picture is empty, so a mask of 'B' will be assumed.

See Also

AddYesNoField

Declaration

```
procedure AddByteField(var ESR : ESrecord; Prompt : string; pRow, pCol : Byte;
                       Picture : string; eRow, eCol : Byte; HelpIndex : Word;
                       ByteLo, ByteHi : Byte; var EditByte : Byte);
```

Purpose

Add a field of type Byte to a data entry screen.

Comments

This routine adds a Byte field to the specified entry screen. Its parameters are essentially identical to those for routines that add fields of other integer types (ShortInt, Integer, Word, LongInt) and for fields of type Char.

ESR is the edit screen to add the Byte field to. Prompt is the message to be displayed to the left (usually) of the edit field. pRow and pCol are the screen coordinates for the Prompt. Picture is the picture mask for the field. Note that this parameter is required for Byte fields. eRow and eCol are the screen coordinates for the edit field. HelpIndex is the index to be passed to the user-defined help routine, if there is one, when <F1> is pressed. ByteLo and ByteHi represent the minimum and maximum values for the Byte being edited. If they are equal, no special range checking is performed. EditByte is the Byte variable to be edited.

Example

```
var Children : Byte;
...
AddByteField(ESR, 'Children', 1, 1, '99', 1, 10, 1, 0, 20, Children);
```

Children is a field of type BCD to be displayed at the top left corner of the screen. Notice that the ByteLo and ByteHi parameters require that the value be in the range 0..20. If these two parameters were equal, the maximum value would be 99, as is indicated by the picture mask.

Declaration

```
procedure AddCharField(var ESR : ESrecord; Prompt : string; pRow, pCol : Byte;
                       Picture : string; eRow, eCol : Byte; HelpIndex : Word;
                       CharLo, CharHi : Char; var EditChar : Char);
```

Purpose

Add a field of type Char to a data entry screen.

Comments

This routine adds a Char field to the specified entry screen. Its parameters are identical to those for AddByteField except that it expects Chars rather than Bytes for the last three parameters. For a discussion of the other parameters, see the section on AddByteField.

See Also

AddByteField

AddChoiceField

Declaration

```
procedure AddChoiceField(var ESR : ESrecord;
                         Prompt : string;
                         pRow, pCol : Byte;
                         Picture : string;
                         eRow, eCol : Byte;
                         HelpIndex : Word;
                         DataSize : Byte;
                         Increment : Pointer;
                         var EditVar);
```

Purpose

Add a multiple choice field to a data entry screen.

Comments

This routine adds a multiple-choice field to the specified entry screen. Its parameters are similar to those for routines that add other types of fields, except for the DataSize and Increment parameters described below.

ESR is the edit screen to add the Byte field to. Prompt is the message to be displayed to the left (usually) of the edit field. pRow and pCol are the screen coordinates for the Prompt. Picture is the picture mask for the field. Note that this parameter is required for multiple-choice fields. eRow and eCol are the screen coordinates for the edit field. HelpIndex is the index to be passed to the user-defined help routine, if there is one, when <F1> is pressed. DataSize is the size of the variable being edited (EditVar); it must be in the range 1-10. Increment is a pointer to a routine that can both convert a variable of the same type as EditVar to a string and can increment or decrement its value. EditVar is the variable to be edited.

See the section on "Multiple-Choice Fields" at the beginning of this chapter for more details, and ENTRY.PAS for an example of a routine that handles incrementing/decrementing chores.

Example

```
type ColorType = (Red, Blue, Yellow);
var  Color : ColorType;
...
{$F+}
procedure IncValue(var Value; ID : Byte; Factor : Integer; var S : string);
begin
  ...
end;
{$F-}
...
AddChoiceField(ESR, 'Color', 1, 1, 'XXXXXX', 1, 7, 1, 1, @IncValue, Color);
```

This example adds a multiple-choice field that will be displayed at the top left corner of the screen. IncValue's job is to adjust Value by the amount indicated by Factor, and return a string representation of the result in S.

Declaration

```
procedure AddCompField(var ESR : ESrecord;
                       Prompt : string;
                       pRow, pCol : Byte;
                       Picture : string;
                       eRow, eCol : Byte;
                       HelpIndex : Word;
                       CompLo, CompHi : Comp;
                       var EditComp : Comp);
```

Purpose

Add a field of type Comp to a data entry screen.

Comments

This routine adds a Comp field to the specified entry screen. Its parameters are identical to those for AddBcdField except that it expects Comps rather than BCD reals for the last three parameters, and it takes no Places parameter, since Comps cannot hold fractional values. For a discussion of the other parameters, and some generally pertinent examples, see the section on AddBcdField.

Note: In order to use AddCompField, you must compile TPENTRY.PAS with the {$N+} compiler directive enabled.

See Also

AddBcdField

AddDateField

Declaration

```
procedure AddDateField(var ESR : ESrecord;
                       Prompt : string;
                       pRow, pCol : Byte;
                       Picture : DateString;
                       eRow, eCol : Byte;
                       HelpIndex : Word;
                       DateLo, DateHi : Date;
                       var EditDate : Date);
```

Purpose

Add an edit field of type Date to a data entry screen.

Comments

This routine adds a Date field to the specified entry screen. Its parameters are essentially identical to those for routines that add fields of integer types.

ESR is the edit screen to add the Date field to. Prompt is the message to be displayed to the left (usually) of the edit field. pRow and pCol are the screen coordinates for the Prompt. Picture is the picture mask for the field. Note that this parameter is required for Date fields. eRow and eCol are the screen coordinates for the edit field. HelpIndex is the index to be passed to the user-defined help routine, if there is one, when <F1> is pressed. DateLo and DateHi represent the minimum and maximum values for the Date being edited. If they are equal, no special range checking is performed, although the date entered must either 1) be a valid date between 1/1/1600 and 12/31/3999 or 2) consist entirely of zeros ('00/00/0000') or spaces (' / / '). EditDate is the Date variable to be edited.

You can suppress the display of 0's in empty date fields by initializing the variable in question to BadDate, and setting both the minimum and maximum values for the field to MinDate (0). This is demonstrated in ENTRY.PAS.

If you use a picture mask such as 'mm/yy' for date fields, a day number of 1 is assumed. You can also use a picture mask such as 'mm/dd'; the global variable DefaultYear (initialized to the current year) specifies the year that is assumed.

See Chapter 3, Section C for more information about date variables.

Example

```
var MyDate : Date;
...
AddDateField(ESR, 'Date', 1, 1, 'mm/dd/yyyy', 1, 6, 1, MinDate, Today, MyDate);
```

This adds a date field to be displayed at the top left corner of the screen. MyDate must be a date between 1/1/1600 and today's date (Today is a function in the TPDATE unit).

See Also

AddDateStField

Today (TPDATE)

Declaration

```
procedure AddDateStField(var ESR : ESrecord;
                         Prompt : string;
                         pRow, pCol : Byte;
                         Picture : DateString;
                         eRow, eCol : Byte;
                         HelpIndex : Word;
                         var EditDate : DateString);
```

Purpose

Add an edit field of type DateString to a data entry screen.

Comments

This routine adds a field of type DateString to the specified edit screen. It is provided for use in cases where dates need to be stored as strings rather than date variables.

The parameters passed to AddDateStField are essentially identical to those for AddStringField except that the optional pointer to a user-defined Validation routine is not necessary. Validation is always handled by a built-in routine that relies on the DateStringToDMY function in the TPDATE unit. See the section on AddStringField for a discussion of the other parameters.

Note that date strings have the same range as date variables: 1/1/1600 to 12/31/3999.

Example

```
var MyDateString : String[10];
...
AddDateStField(ESR, 'Date', 1, 1, 'mm/dd/yyyy', 1, 6, 1, MyDateString);
```

This adds a date string field to be displayed at the top left corner of the screen.

See Also

AddDateField
AddStringField
DateStringToDMY (TPDATE)

AddDblField / AddExtField

Declaration

```
procedure AddDblField(var ESR : ESrecord; Prompt : string; pRow, pCol : Byte;
                      Picture : string; eRow, eCol : Byte; HelpIndex : Word;
                      DblLo, DblHi : Double; Places : Byte;
                      var EditDbl : Double);
```

Purpose

Add a field of type Double to a data entry screen.

Comments

This routine adds a Double field to the specified entry screen. Its parameters are identical to those for AddBcdField except that it expects Doubles rather than BCD reals for three of the last four parameters. For a discussion of the other parameters, and some generally pertinent examples, see the section on AddBcdField.

Note: In order to use AddDblField, you must compile TPENTRY.PAS with the {$N+} compiler directive enabled.

See Also

AddBcdField

Declaration

```
procedure AddExtField(var ESR : ESrecord; Prompt : string; pRow, pCol : Byte;
                      Picture : string; eRow, eCol : Byte; HelpIndex : Word;
                      ExtLo, ExtHi : Extended; Places : Byte;
                      var EditExt : Extended);
```

Purpose

Add a field of type Extended to a data entry screen.

Comments

This routine adds an Extended field to the specified entry screen. Its parameters are identical to those for AddBcdField except that it expects Extendeds rather than BCD reals for three of the last four parameters. For a discussion of the other parameters, and some generally pertinent examples, see the section on AddBcdField.

Note: In order to use AddExtField, you must compile TPENTRY.PAS with the {$N+} compiler directive enabled.

See Also

AddBcdField

Declaration

```
procedure AddIntField(var ESR : ESrecord; Prompt : string; pRow, pCol : Byte;
                      Picture : string; eRow, eCol : Byte; HelpIndex : Word;
                      IntLo, IntHi : Integer; var EditInt : Integer);
```

Purpose

Add a field of type Integer to a data entry screen.

Comments

This routine adds an Integer field to the specified entry screen. Its parameters are identical to those for AddByteField except that it expects Integers rather than Bytes for the last three parameters. For a discussion of the other parameters, see the section on AddByteField.

See Also

AddByteField

Declaration

```
procedure AddLongField(var ESR : ESrecord; Prompt : string;
                       pRow, pCol : Byte; Picture : string;
                       eRow, eCol : Byte; HelpIndex : Word;
                       LongLo, LongHi : LongInt; var EditLong : LongInt);
```

Purpose

Add a field of type LongInt to a data entry screen.

Comments

This routine adds a LongInt field to the specified entry screen. Its parameters are identical to those for AddByteField except that it expects LongInts rather than Bytes for the last three parameters. For a discussion of the other parameters, see the section on AddByteField.

See Also

AddByteField

AddNestedField

Declaration

```
procedure AddNestedField(var ESR : ESrecord;
                         Prompt : string;
                         pRow, pCol : Byte;
                         Picture : string;
                         eRow, eCol,
                         EditLen : Byte;
                         HelpIndex : Word);
```

Purpose

Add a field that is associated with a nested data entry screen.

Comments

This routine allows you to add a field to the specified edit screen that forms a link to a secondary or "nested" data entry screen. A field of this type has no data associated with it, although it does have a dummy edit field for cosmetic reasons. If the cursor is positioned on an edit field of this type and the user presses <Enter>, the edit screen manager returns control to the main program, passing back a special exit command code, ESnested. It is then up to the user program to invoke the secondary edit screen. See ENTRY.PAS for an example of how to implement nested entry screens.

ESR is the edit screen to add the field to. Prompt is the message to be displayed to the left (usually) of the edit field. pRow and pCol are the screen coordinates for the Prompt. Picture is the optional picture mask for the field. If no picture mask is specified, AddStringField will assume an implied picture mask of length EditLen consisting entirely of 'X'. If it is not empty, Picture must contain at least one formatting character ('X', '!', 'A', etc.) so that there will be a place to put the cursor when it is on this field. eRow and eCol are the screen coordinates for the edit field. EditLen is the width of the editing field. It is optional unless Picture is empty. HelpIndex is the index to be passed to the user-defined help routine, if there is one, when <F1> is pressed.

Example

```
AddNestedField(ESR, 'Employment history', 1, 1, '', 1, 14, 1, 1);
```

In this example, Employment history is a field that presumably leads to a secondary edit screen containing fields related to the general category of Employment history (past employers, places of employment, years worked).

Declaration

```
procedure AddRealField(var ESR : ESrecord; Prompt : string; pRow, pCol : Byte;
                       Picture : string; eRow, eCol : Byte; HelpIndex : Word;
                       RealLo, RealHi : Real; Places : Byte;
                       var EditReal : Real);
```

Purpose

Add a field of type Real to a data entry screen.

Comments

This routine adds a Real field to the specified entry screen. Its parameters are identical to those for AddBcdField except that it expects ordinary six-byte Reals rather than BCD reals for three of the last four parameters. For a discussion of the other parameters, and some generally pertinent examples, see the section on AddBcdField.

See Also

AddBcdField

Declaration

```
procedure AddSglField(var ESR : ESrecord; Prompt : string; pRow, pCol : Byte;
                      Picture : string; eRow, eCol : Byte; HelpIndex : Word;
                      SglLo, SglHi : Single; Places : Byte;
                      var EditSgl : Single);
```

Purpose

Add a field of type Single to a data entry screen.

Comments

This routine adds a Single field to the specified entry screen. Its parameters are identical to those for AddBcdField except that it expects Singles rather than BCD reals for three of the last four parameters. For a discussion of the other parameters, and some generally pertinent examples, see the section on AddBcdField.

Note: In order to use AddSglField, you must compile TPENTRY.PAS with the {$N+} compiler directive enabled.

See Also

AddBcdField

AddShortField

Declaration

```
procedure AddShortField(var ESR : ESrecord;
                        Prompt : string;
                        pRow, pCol : Byte;
                        Picture : string;
                        eRow, eCol : Byte;
                        HelpIndex : Word;
                        ShortLo, ShortHi : ShortInt;
                        var EditShort : ShortInt);
```

Purpose

Add a field of type ShortInt to a data entry screen.

Comments

This routine adds a ShortInt field to the specified entry screen. Its parameters are identical to those for AddByteField except that it expects ShortInts rather than Bytes for the last three parameters. For a discussion of the other parameters, see the section on AddByteField.

See Also

AddByteField

Declaration

```
procedure AddStringField(var ESR : ESrecord;
                         Prompt : string;
                         pRow, pCol : Byte;
                         Picture : string;
                         eRow, eCol,
                         EditLen : Byte;
                         HelpIndex : Word;
                         Validation : Pointer;
                         var EditSt : string);
```

Purpose

Add a field of type string to a data entry screen.

Comments

This routine adds a string field to the specified entry screen. The chief differences between it and most of the other field addition routines are 1) it accepts as a parameter a pointer to a special validation routine, and 2) it provides a means of editing strings longer than the width of the edit field.

ESR is the edit screen to add the string field to. Prompt is the message to be displayed to the left (usually) of the edit field. pRow and pCol are the screen coordinates for the Prompt. Picture is the optional picture mask for the field. If no picture mask is specified, AddStringField will assume an implied picture mask of length EditLen consisting entirely of 'X' (allows any character to be entered). eRow and eCol are the screen coordinates for the edit field. EditLen is the width of the editing field. If EditLen < Length(Picture), the field will be scrolled horizontally as necessary while editing. HelpIndex is the index to be passed to the user-defined help routine, if there is one, when <F1> is pressed. If not nil, Validation must point to a built-in or user-written validation routine. EditSt is the string to be edited. It must be equal in size to EditLen or Length(Picture), whichever is larger.

Examples

```
AddStringField(ESR, 'Notes', 19, 18, CharStr('X', 254), 19, 25, 46, 16, nil,
  Scrap.Notes);
```

Here Scrap.Notes is a String[254], as is indicated by the length of the picture mask. Since the EditLen parameter is 46, the field will have to scroll horizontally while editing if the string becomes longer than 46 characters.

```
AddStringField(ESR, 'State', 10, 18, 'AA', 10, 25, 2, 4, @ValidateNoBlanks,
  Scrap.State);
```

Scrap.State is a String[2], as is indicated both by the length of the picture mask and by the EditLen parameter. ValidateNoBlanks is a built-in validation routine described later in the chapter.

See Also

AddArrayField

ValidateNoBlanks

AddTimeField

Declaration

```
procedure AddTimeField(var ESR : ESrecord;
                       Prompt : string;
                       pRow, pCol : Byte;
                       Picture : DateString;
                       eRow, eCol : Byte;
                       HelpIndex : Word;
                       TimeLo, TimeHi : Time;
                       var EditTime : Time);
```

Purpose

Add an edit field of type Time to a data entry screen.

Comments

This routine adds a Time field to the specified entry screen. Its parameters are essentially identical to those for routines that add fields of integer types.

ESR is the edit screen to add the Time field to. Prompt is the message to be displayed to the left (usually) of the edit field. pRow and pCol are the screen coordinates for the Prompt. Picture is the picture mask for the field. Note that this parameter is required for Time fields. eRow and eCol are the screen coordinates for the edit field. HelpIndex is the index to be passed to the user-defined help routine, if there is one, when <F1> is pressed. TimeLo and TimeHi represent the minimum and maximum values for the Time being edited. If they are equal, no special range checking is performed. EditTime is the Time variable to be edited.

You can suppress the display of 0's in empty time fields by initializing the variable in question to BadTime, and setting both the minimum and maximum values for the field to MinTime (0).

See Chapter 3, Section C for more information about time variables.

Example

```
var MyTime : Time;
...
AddTimeField(ESR, 'Time', 1, 1, 'hh:mm:ss te', 1, 6, 1, MinTime, MaxTime,
  MyTime);
```

This adds a time field to be displayed at the top left corner of the screen. Notice that the time is displayed and entered in am/pm format.

Declaration

```
procedure AddUserField(var ESR : ESrecord;
                       Prompt : string;
                       pRow, pCol : Byte;
                       Picture : string;
                       eRow, eCol,
                       EditLen : Byte;
                       HelpIndex : Word;
                       Validation, Conversion : Pointer;
                       UserPtr1, UserPtr2 : Pointer;
                       var EditVar);
```

Purpose

Add a field of a user-defined type to a data entry screen.

Comments

This routine adds a field of a user-defined type to the specified entry screen. It is provided as a convenience for advanced programmers who have written their own special conversion and validation routines to deal with data types not supported by TPENTRY.

ESR is the edit screen to add the field to. Prompt is the message to be displayed to the left (usually) of the edit field. pRow and pCol are the screen coordinates for the Prompt. Picture is the optional picture mask for the field. If no picture mask is specified, AddUserField will assume an implied picture mask of length EditLen consisting entirely of 'X' (allows any character to be entered). eRow and eCol are the screen coordinates for the edit field. EditLen is the width of the editing field. If EditLen < Length(Picture), the field will be scrolled horizontally as necessary while editing. HelpIndex is the index to be passed to the user-defined help routine, if there is one, when <F1> is pressed. Validation must point to a built-in or user-written validation routine, and Conversion must point to a user-written conversion routine. Both these variables are required for user-defined fields. UserPtr1 and UserPtr2 may be used to store pointers to data needed by the conversion or validation routine. EditVar is the variable to be edited.

See the section on "User-defined Types" at the beginning of this chapter for more details.

Example

```
var MyVar : MyDataType;
...
AddUserField(ESR, 'Something', 1, 1, '', 1, 11, 20, 1, @MyValidationRoutine,
  @MyConversionRoutine, nil, nil, MyVar);
```

This example adds a field of type MyDataType to the edit screen. MyValidationRoutine and MyConversionRoutine are user-written routines that handle validation and conversion of fields of type MyDataType.

AddWordField / AddYesNoField

Declaration

```
procedure AddWordField(var ESR : ESrecord; Prompt : string; pRow, pCol : Byte;
                       Picture : string; eRow, eCol : Byte; HelpIndex : Word;
                       WordLo, WordHi : Word; var EditWord : Word);
```

Purpose

Add a field of type Word to a data entry screen.

Comments

This routine adds a Word field to the specified entry screen. Its parameters are identical to those for AddByteField except that it expects Words rather than Bytes for the last three parameters. For a discussion of the other parameters, see the section on AddByteField.

See Also

AddByteField

Declaration

```
procedure AddYesNoField(var ESR : ESrecord; Prompt : string; pRow, pCol : Byte;
                        Picture : string; eRow, eCol : Byte; HelpIndex : Word;
                        var EditYesNo : Boolean);
```

Purpose

Add a field of type yes/no to a data entry screen.

Comments

This routine adds a YesNo field to the specified entry screen. Its parameters are essentially identical to those for AddBooleanField. For a discussion of the other parameters, see the section on AddBooleanField. The one difference between the two procedures is that a picture mask of 'Y' is assumed if Picture is empty.

See Also

AddBooleanField

ChangeFieldAttr / ChangePromptAttr

Declaration

```
procedure ChangeFieldAttr(var ESR : ESrecord; FieldID : Word; A : Byte);
```

Purpose

Change the field attribute for the specified field.

Comments

This routine allows you to change the field attribute for a field after it has been added to the entry screen, perhaps while within a post-edit routine.

ESR is the entry screen, FieldID is the field to change, and A is the new attribute. See "Hidden and Semi-Hidden Fields" in the overview of this section for a discussion of what this routine can be used for. Note that a call to this procedure changes the attribute used to display the field, even if it is protected.

Declaration

```
procedure ChangePromptAttr(var ESR : ESrecord; FieldID : Word; A : Byte);
```

Purpose

Change the prompt attribute for the specified field.

Comments

This routine allows you to change the prompt attribute for a field after it has been added to the entry screen, perhaps while within a post-edit routine.

See "Hidden and Semi-Hidden Fields" in the overview of this section for a discussion of what this routine can be used for. Note that a call to this procedure changes the attribute used to display the prompt, even if the field is protected.

ChangeProtection

Declaration

```
procedure ChangeProtection(var ESR : ESrecord; FieldID : Byte; OnOff : Boolean);
```

Purpose

Modify the protection status of a field.

Comments

Ordinarily, a field is designated as protected or unprotected at the time it is added to an edit screen, and it retains its protection status for as long as the edit screen exists. In some cases, however, you may need to protect or unprotect a field later in response to something the user has entered. For example, you might want to mark a field as protected as soon as the user has entered valid data into it. Or you might want to unprotect a field that was initially protected as soon as the user has entered valid data into another field. Whatever your reasons, ChangeProtection gives you the means to do it.

Example

```
ChangeProtection(ESR, 5, On);
```

Mark the sixth field in the designated edit screen (field ID numbers start with 0) as protected.

See Also

SetProtection

SetWrapMode

ChangeProtectionFast / ChangeRequired

Declaration

```
procedure ChangeProtectionFast(var ESR : ESrecord; FieldID : Word;
                               OnOff : Boolean);
```

Purpose

Modify the protection status of a field.

Comments

This routine performs the same basic function as ChangeProtection, except that it does not call ResetEntryScreenFlags. It is intended to be used in place of ChangeProtection in cases where you are changing the protection status of multiple fields at a time. Once all the calls to ChangeProtectionFast are made, you *must* call ResetEntryScreenFlags *if* this routine is being called from within a post-edit routine.

Example

See BEST.PAS for an example of a case where ChangeProtectionFast is needed.

See Also

ResetEntryScreenFlags

Declaration

```
procedure ChangeRequired(var ESR : ESrecord; FieldID : Word;
                         OnOff : Boolean);
```

Purpose

Modify the required status of a field after it has been added.

Comments

This routine is identical to ChangeProtection, except that it changes a field's required status rather than its protection status.

ChangeStringAttr / ChangeValidation

Declaration

```
procedure ChangeStringAttr(var ESR : ESrecord; FieldID : Word; A : Byte);
```

Purpose

Change the string attribute for the specified field.

Comments

This routine allows you to change the string attribute for a field after it has been added to the entry screen, perhaps while within a post-edit routine.

See "Hidden and Semi-Hidden Fields" in the overview of this section for a discussion of what this routine can be used for.

See Also

ChangeFieldAttr

Declaration

```
procedure ChangeValidation(var ESR  : ESrecord; FieldID  : Word;
                           Validate : Pointer);
```

Purpose

Change the validation routine for the specified field.

Comments

This routine allows you to alter the validation routine for any field in an entry screen. ESR is the entry screen. FieldId is the ID number for the field. Validate is the address of the validation routine. You might use ChangeValidation to specify a special validation routine for a field of type Real, for example. Or you might want to use two different validation routines for a given field, with the choice of routines depending on whether validation needs to be strict or relaxed in a given situation.

Example

```
ChangeValidation(ESR, 5, @SpecialValidationRoutine);
```

Change the validation routine pointer for the sixth field in the designated entry screen to point to SpecialValidationRoutine.

ClearEntryWindow / CurrentFieldModified

Declaration

```
procedure ClearEntryWindow(var ESR : ESrecord);
```

Purpose

Clear the window for the entry screen.

Comments

This clears the data entry window associated with the specified data entry screen. If no window coordinates have been set (using SetEntryWindow), the entire screen will be cleared.

Example

```
SetEntryWindow(ESR, 1, 5, 80, 20, False, TextAttr, TextAttr);
...
ClearEntryWindow(ESR);
```

This would clear columns 1-80 of the screen from row 5 to row 20.

See Also

SetEntryWindow

Declaration

```
function CurrentFieldModified(var ESR : ESrecord) : Boolean;
```

Purpose

Return True if the current field was modified.

Comments

This routine is intended to be used only within a post-edit routine to determine whether the current field was actually modified by the user. It is especially useful in applications where the post-edit routine must load data from disk based on the value of a particular field, and you don't want to reread the same data unnecessarily.

Note that, if the current field is a multiple choice field, CurrentFieldModified will return True only if the user has actually left the field. (The multiple choice editor calls the post-edit routine each time the user increments or decrements the value of the field. This allows calculated fields that depend on the current field to be updated constantly.)

See Also

EvaluateEScommand

DisableEntryMouse / DisposeEditScreen

Declaration

```
procedure DisableEntryMouse;
```

Purpose

Disable mouse support in TPENTRY.

Comments

This routine deactivates mouse support in TPENTRY. Besides clearing the internal flag set by EnableEntryMouse, it deactivates the mouse event handler in TPMOUSE and sets EntryKeyPtr to point to ReadKeyWord (TPCRT).

See Also

EnableEntryMouse

Declaration

```
procedure DisposeEditScreen(var ESR : ESrecord);
```

Purpose

Deallocate all field records associated with ESR.

Comments

This is a "cleanup routine". It disposes of all memory that was allocated dynamically when fields were being added to an edit screen, and erases all data in the ESrecord.

In many applications, particularly those containing only a single edit screen, this procedure never needs to be called. It is most likely to be needed in larger programs containing multiple edit screens, where the dynamic memory used by the edit screen needs to be recovered after each editing session.

Example

```
procedure MyEditScreen;
var ESR : ESrecord;
begin
  InitESrecord(ESR);
  {add fields to the edit screen}
  repeat
    ExitCommand := EditScreen(ESR, 0);
    {process exit command}
  until Done;
  DisposeEditScreen(ESR);
end;
```

A typical use of DisposeEditScreen: the ESrecord is local to a procedure which might be called multiple times in the program; memory is allocated at the beginning of the procedure when fields are being added; and the memory needs to be recovered at the end of the procedure.

See Also

InitESrecord

Declaration

```
procedure DrawEditScreen(var ESR : ESrecord);
```

Purpose

Redraw an entire edit screen.

Comments

This routine calls DrawField repeatedly in order to display each field in an edit screen. Since EditScreen does this automatically when editing begins, use of DrawEditScreen should rarely be necessary. It is most useful, however, in cases where you want to display data without allowing it to be edited. See the DisplayRecord procedure in the demonstration program PTRPT.PAS for an example of this use of DrawEditScreen.

See Also

DrawField

Declaration

```
procedure DrawField(var ESR : ESrecord; FieldID : Byte);
```

Purpose

Draw the edit field corresponding to FieldID.

Comments

This routine redraws the field corresponding to FieldID in the designated edit screen. The most common use for it is in a post-edit routine that handles calculated fields, where a field needs to be redisplayed after a a new value has just been calculated. For an example of this use of DrawField, see ENTRY.PAS.

Example

```
DrawField(ESR, ESR.CurrentID);
```

Redraws the current field in the designated edit screen.

See Also

DrawEditScreen

SetPostEditPtr

EditScreen

Declaration

```
function EditScreen(var ESR : ESrecord; StartField : Word;
                    ReadOnly : Boolean) : EStype;
```

Purpose

Edit fields until an exit command is issued.

Comments

This is the central routine in TPENTRY and contains the core of the edit screen manager. It is typically called from within a repeat..until loop, in which the edit screen manager and the field editors handle all the editing and cursor movement commands while the user program handles the exit commands, returned in the function result of EditScreen.

In most cases, the user program must concern itself with only four exit commands: ESnextRec, which indicates that the user wants to edit the next record in a series; ESprevRec, which indicates that he wants to edit the previous record; ESdone, which indicates that he is finished editing; and ESquit, which indicates that he wants to abort. Note that ESnextRec and ESprevRec are meaningful only in database-type applications in which there are multiple records to be edited. The only other valid exit commands are the user-defined exit commands, ESuser0..ESuser9. In cases where an edit screen has not been initialized properly, or contains no unprotected fields, a result of ESnone is also possible, however.

The StartField parameter indicates the initial position of the cursor within the edit screen. The most common values are 0, meaning the first field in the edit screen, and ESR.CurrentID (see the examples below). If the field specified is a protected field, the edit screen manager will place the cursor on the first unprotected field following the designated field instead. The ReadOnly parameter is used to select read-only mode, in which the user can move from field to field and record to record, but cannot execute any editing commands.

For an example of how to use EditScreen, as well as how to implement user-defined exit commands, see ENTRY.PAS.

Examples

```
ExitCommand := EditScreen(ESR, 0, False);
```

Begin an editing session, and place the cursor on the first (unprotected) field in the edit screen.

```
ExitCommand := EditScreen(ESR, ESR.CurrentID, True);
```

Begin (or continue) an editing session with the cursor placed on the same field that it was on the last time that EditScreen was called. To see what effect this has, load the demonstration program ENTRY.EXE, press <CtrlPgDn>, then <PgDn>. The third parameter selects read-only mode.

Declaration

```
procedure EditString(Prompt : string;
                     Row, Col : Byte;
                     EditLen : Byte;
                     Picture : string;
                     HelpIndex : Word;
                     var Escaped : Boolean;
                     var EditSt : string);
```

Purpose

Use one of the field editors to edit a string.

Comments

This routine allows two of the field editors (the string editor and the numeric editor) to be used as line editors, much like the ReadString procedure in TPEDIT. The chief difference between this routine and ReadString is that it allows you to specify a picture mask.

There *are* other differences, however, for in some respects a call to EditString is much like a call to AddStringField: all special editing options must be selected by making calls to the appropriate SetOption routines beforehand. For example, to set the colors that the prompt is displayed in, you would call SetPromptAttr, and to select use of the numeric editor, you would call SetNumeric. (Note, however, that if Picture contains '$' or ',', numeric mode is selected automatically unless IncludeNumeric is unDEFINEd, in which case the program will be aborted.)

Another difference between EditString and ReadString to keep in mind is that the string being edited will be padded to its full length with blanks unless (a) the blank trimming option is on, as it is by default, or (b) the numeric editor is being used. In numeric mode, any special Picture characters (such as '$' or ',') are also stripped from the string before it is returned. If you are using EditString to edit numbers, you must therefore convert the number to a string beforehand, and convert it back to a number afterwards. The numeric string passed in initially (EditSt) should not contain any special characters, just a raw number such as '-10000.12' or an empty string. If the string editor is used, EditSt should either be empty or be pre-formatted. For example, if the picture mask is '(999) 999-9999', EditSt might contain '( ) - '.

The parameters to EditString are generally similar to those for AddStringField. Prompt is the message to be displayed to the left of the edit field. Row and Col are the screen coordinates for the Prompt. Picture is the optional picture mask for the field. If no picture mask is specified, EditString will assume an implied picture mask of length EditLen consisting entirely of 'X' (allows any character to be entered). EditLen is the width of the editing field, which begins just beyond the end of the Prompt. If EditLen < Length(Picture), the field will be scrolled horizontally as necessary while editing. HelpIndex is the index to be passed to the user-defined help routine, if there is one, when <F1> is pressed. Escaped will be set to True on exit if the ESquit command (normally assigned to <Esc>) was given. EditSt is the string to be edited. It must be equal in size to EditLen or Length(Picture), whichever is larger.

EditString

Examples

```
FName := '';
EditString('Filename: ', 1, 1, 64, '', 0, Escaped, FName);
```

Use the string editor to obtain a filename.

```
S := '';
SetNumeric(On);
EditString('Enter a number: ', 1, 1, 5, '99999', 0, Escaped, S);
SetNumeric(Off);
```

Use the numeric editor to obtain a number in the range 0-99,999.

See Also

AddStringField

ReadString (TPEDIT)

EnableEntryMouse

Declaration

```
procedure EnableEntryMouse;
```

Purpose

Enable mouse support in TPENTRY.

Comments

This routine activates mouse support in TPENTRY. In addition to setting an internal flag for use by other routines in TPENTRY, it activates the mouse event handler in TPMOUSE and sets EntryKeyPtr to point to the ReadKeyOrButton routine in TPMOUSE. See the introduction to TPMOUSE in section 2.G for more details.

See Also

DisableEntryMouse

EvaluateEScommand

Declaration

```
function EvaluateEScommand(var ESR : ESrecord; ESC : EStype) : FieldRecPtr;
```

Purpose

Given a command, return a pointer to the field the cursor will move to.

Comments

This routine is intended to be called only from within a post-edit routine. Given an entry screen record (ESR) and the code for the last command entered by the user (stored in the global variable LastEntryCommand), it calculates where EntryScreen will move the cursor after the post-edit routine returns control. If the cursor will not be moved at all, the function result is ESR.CurrentField. Otherwise, it points to the field to move to.

EvaluateEScommand is especially useful in cases where a post-edit routine needs to load additional data from disk after the user enters a value in a particular field. For example, after the user enters a customer ID number, the post-edit routine might need to load the rest of the known data concerning that customer (name, address, etc.).

Example

```
{$F+}
procedure PostEditRoutine(var ESR : ESrecord);
var
  FRP : FieldRecPtr;
begin
  if (ESR.CurrentID = FieldOfInterest) then
    if CurrentFieldModified(ESR) then begin
      FRP := EvaluateEScommand(ESR, LastEntryCommand);
      case FRP^.FieldID of
        5..8 :
          begin
            {load some more data}
            ...
            DrawEditScreen(ESR);
          end;
      end;
    end;
end;
{$F-}
```

This sample post-edit routine checks first to see if the cursor is on a field of interest, then insures that the user actually modified the field, then calls EvaluateEScommand. If the cursor is going to be moved into a particular region of the entry screen (fields 5-8), it loads new data from disk and redraws the entry screen.

Declaration

```
procedure InitESrecord(var ESR : ESrecord);
```

Purpose

Initialize an ESrecord.

Comments

This procedure must be called before adding any fields to an edit screen. It initializes all the variables in the ESrecord with their default values, and inserts a "signature" that allows other routines to determine whether or not the ESrecord has been initialized properly. If it hasn't, and you try to add a field to it, the program will be aborted.

Examples

```
SetPreEditPtr(ESR, @DisplayHelpMessage);
InitESrecord(ESR);
```

Incorrect! ESR *must* be initialized before the call to SetPreEditPtr.

```
InitESrecord(ESR);
SetPreEditPtr(ESR, @DisplayHelpMessage);
```

Correct.

InternationalCurrency

Declaration

```
function InternationalCurrency(FormChar : Char; MaxDigits : Byte;
                               FloatIfPossible : Boolean;
                               AddCommas : Boolean) : string;
```

Purpose

Return a picture mask for a currency string, based on DOS's country info.

Comments

This routine uses DOS's Get Country Information service in order to determine not only the proper values for DecimalChar, CommaChar, CurrencyLtStr, and CurrencyRtStr, but also how to produce a picture mask that is appropriate for the current country's currency.

FormChar is the formatting character to use (usually '9' or '#'). MaxDigits is the number of digits that may appear to the left of the decimal point (DOS indicates how many there should be to the right). FloatIfPossible indicates whether '$' or 'c' should be used if the currency symbol goes on the left. AddCommas indicates whether discretionary commas should appear in the picture mask.

When using this routine, you need to keep in mind that the string returned might be as large as 8+MaxDigits characters long, not counting any space set aside for commas (Pred(MaxDigits) div 3 characters). This is because a currency symbol can be as long as 5 characters, and there will usually be a decimal point with two digits on its right as well. You should also bear this in mind when laying out your entry screens: leave room for the largest picture mask that you might possibly receive.

Note that the information provided by DOS 2.x is minimal compared to that available under DOS 3.x and higher. We therefore recommend that this function be used with caution in programs that must run under DOS 2. Where information is lacking, US conventions are followed.

Examples

```
InternationalCurrency('9', 6, True, True)
```

Returns '$999,999.99' for US (DecimalChar = '.', CommaChar = ',', CurrencyLtStr = '$', CurrencyRtStr = ''), '999,999CCCC' for Italy (DecimalChar unused, ColonChar = '.', CurrencyLtStr = '', CurrencyRtStr = 'Lit.').

```
InternationalCurrency('#', 4, False, False)
```

Returns 'c####.##' for US, 'cccc####.##' for Sweden (DecimalChar = ',', CommaChar = '.', CurrencyLtStr = 'SEK ').

See Also

InternationalDate (TPDATE) InternationalTime (TPDATE)

Declaration

```
procedure ResetEntryScreenFlags(var ESR : ESrecord);
```

Purpose

Sets/clears a series of flags used internally by TPENTRY.

Comments

This routine is intended primarily for internal use, but it is interfaced for use by the programmer in cases where ChangeProtectionFast is being called from within a post-edit routine. This routine does not need to be called except in that one very specific case. In all other cases where it needs to be called, TPENTRY calls it automatically.

This routine follows the linked list of fields that make up an entry screen, setting flags to mark those fields that are on the first or last row of the entry screen. These flags simplify and speed up certain operations. There is a brief discussion of such cases in "Hidden and Semi-Hidden Fields" in the overview of this section.

Example

For an example of a case where both ResetEntryScreenFlags and ChangeProtectionFast are needed, see BEST.PAS.

Declaration

```
procedure RingBell;
```

Purpose

Ring the bell.

Comments

RingBell is called if a validation error occurs and no user-defined error handler is installed, or if an invalid character is entered. It is interfaced so that other parts of a program may handle errors in the same manner as TPENTRY does.

Example

```
if ErrorOccurred then begin
  WriteLn(ErrorMessage);
  RingBell;
end;
```

Display an error message and ring the bell.

See Also

SetBeepOnError SetBell

SetAllFieldLinks

Declaration

```
procedure SetAllFieldLinks(var ESR : ESrecord; var LinksMap);
```

Purpose

Set the forward and backward links for all fields in an entry screen.

Comments

This routine allows you to modify the order in which fields are "visited" as you move through an entry screen by specifying an explicit "forward link" for each field. (The forward link is the field to move the cursor to when <Enter> is pressed.)

ESR is the entry screen whose forward links are to be modified. The variable passed as the LinksMap parameter should be an 'array[0..n] of Word', where n is the ID number of the last field in the entry screen. Each element in the array specifies the forward link for that field: LinksMap[0] has the forward link for field 0, LinksMap[1] has the forward link for field 1, and so on. Note that, given this list of forward links, SetAllFieldLinks automatically calculates the appropriate backward links to insure logical behavior when the cursor is moved backward (with <ShiftTab>, for example).

If a particular field is protected, set its forward link to 'BadFieldId' (= $FFFF) *unless* the field will later be unprotected (by a post-edit routine, perhaps). Note that EditScreen ignores any forward or backward link that points to a protected field and reverts to the default behavior of moving the cursor to the next/previous unprotected field added to the list.

When setting the backward links for all the fields, SetAllFieldLinks assumes that each unprotected field in the entry screen is named as the forward link for one of the other fields, and that only one field names a given field as its forward link. More specifically, when setting the backward link for field N, it scans the list of forward links looking for another field that points to N. The first one that it finds becomes the backward link for field N. If none is found, the backward link is set to BadFieldID, indicating that EditScreen should apply its usual set of rules for determining where to move the cursor. In general, then, it is best to use SetAllFieldLinks to establish a perfect loop linking all unprotected fields in the entry screen together (see the example below), then use SetFieldLinks to make any minor changes to individual links (to prevent wrapping at the edges of the entry screen, for example, which can be done by setting a field's forward or backward link to point to itself).

Calls to SetAllFieldLinks have no effect on the behavior of the <Up> and <Down> commands. Those commands always move the cursor to the field immediately above or below the current field. If you want <Up> and <Down> to move the cursor to the next/previous field, you can reassign them to the ESprevField and ESnextField commands, respectively, using the AddEntryCommand routine described in Appendix B.

Example

```
const LinksMap : array[0..5] of Word =
  {field: 0  1  2  3  4  5}
         (1, 3, 0, 4, 2, BadFieldId);
...
SetAllFieldLinks(ESR, LinksMap);
...
ExitCommand := EditScreen(ESR, 2, False);
```

The entry screen is set up such that the cursor starts at field 2, then moves to fields 0, 1, 3, 4 (and then back to 2), in that order. Field 5 is a protected field, so no forward link is needed for it.

SetAutoAdvance

Declaration

```
procedure SetAutoAdvance(On : Boolean);
```

Purpose

Turn the AutoAdvance option on or off.

Comments

The state of the AutoAdvance option for a given field determines what happens in several common situations. For example: 1) The cursor is at the beginning of a field and the user presses <Left>: if AutoAdvance is on, as it is by default, the cursor would be moved to the field just to the left of the current field; if it's off, nothing happens. 2) The current field, which holds a maximum of five characters, now contains four characters and the user has just entered a fifth: if AutoAdvance is on, the cursor would be advanced to the next field; if not, the user would have to press <Enter> to advance it. 3) The cursor is at the end of the current field, and the user presses <CtrlRight> to move to the next word to the right: if AutoAdvance is on, the cursor would be moved to the beginning of the first word in the next field to the right; if not, nothing would happen.

It is possible to turn AutoAdvance on for entry screens containing a single field. Note, however, that single-field entry screens are still treated as special cases: cursor movement commands such as <Left>, <Right>, <CtrlLeft>, <CtrlRight>, <Tab>, and <ShiftTab> cannot be used to exit from a field as they can in multi-field entry screens. Note too that the EditString routine, which creates a single-field entry screen, does force AutoAdvance off.

Example

```
SetAutoAdvance(Off);
AddStringField(ESR, 'Notes', 19, 18, CharStr('X', 254), 19, 25, 46, 16, nil,
  Scrap.Notes);
SetAutoAdvance(On);
```

Disables AutoAdvance for the Notes field (the string can be longer than the edit field, making it difficult to tell where the string begins and ends).

Declaration

```
procedure SetAutoNumLock(On : Boolean);
```

Purpose

Turn the AutoNumLock option on or off.

Comments

This procedure allows you to activate an option which causes the field editor to automatically activate NumLock each time a particular field is edited, and restore it to its previous state afterward.

Example

```
SetAutoNumLock(On);
AddRealField(ESR, 'Hourly wage', 16, 12, '$99.99', 16, 25, 12, 0, 999.99, 0,
  Scrap.Wage);
SetAutoNumLock(Off);
```

In this case, AutoNumLock is enabled for the Wage field and disabled for all other fields.

See Also

SetNumeric

Declaration

```
procedure SetBeepOnError(var ESR : ESrecord; On : Boolean);
```

Purpose

Requests a beep when an illegal character is entered.

Comments

Ordinarily, when the user tries to enter an illegal character into a field (illegal, that is, according to the picture mask for the field) or a key like <F5> that has no particular function, the field editors will ignore the keystroke. By calling SetBeepOnError, however, you can request that a short beep (or bell) be sounded to notify the user of the error. You can control the sound that is emitted by calling the SetBell procedure. See ENTRY.PAS for an example of how to use SetBeepOnError.

Example

```
SetBeepOnError(ESR, On);
```

Beep when the user tries to enter an illegal character.

See Also

SetBell

SetBell / SetClearFirstChar

Declaration

```
procedure SetBell(Pitch, Duration : Word);
```

Purpose

Set pitch and duration for bell.

Comments

This procedure allows you to control the sound generated by calls to RingBell. The values passed to SetBell are stored in the global variables BellPitch and BellDuration. Their default values are 700 (hertz) and 100 (milliseconds) respectively.

Example

```
SetBell(500, 250);
```

Requests a pitch of 500 hertz and a duration of 1/4 of a second.

See Also

RingBell

Declaration

```
procedure SetClearFirstChar(On : Boolean);
```

Purpose

Turn ClearFirstChar option on or off.

Comments

Ordinarily, when the cursor is first moved to a field that already contains text and the user starts entering characters, the new text is inserted into the old text (if insert mode is on) or overwrites the old text (if it isn't). SetClearFirstChar allows you to specify that, if the first character entered is an ASCII character (as opposed to an editing command), the field will be cleared and the new character inserted at the beginning of the field.

Example

```
SetClearFirstChar(On);
AddStringField(ESR, 'Address', 8, 16, '', 8, 25, 30, 2, nil, Scrap.Address);
SetClearFirstChar(Off);
```

In this example, the ClearFirstChar option is on for the Address field, and off for all other fields.

Declaration

```
procedure SetCtrlAttr(A : Byte);
```

Purpose

Set attribute for control characters.

Comments

This routine lets you set the attribute used to display control characters embedded in strings being edited. If this attribute is the same as the string attribute (set with SetStringAttr), control characters may not be entered. If it is different, control characters may be entered using the <CtrlP> command (ESctrlChar), and they will be displayed as upper case alphabetics (e.g., ^U as 'U') in the designated attribute. The value passed to SetCtrlAttr is stored in the global variable ESctrlAttr.

Example

```
SetStringAttr($7);
SetCtrlAttr($F);
```

Display strings being edited in light gray on black, control characters in white on black.

See Also

SetFieldAttr
SetPromptAttr
SetStringAttr

Declaration

```
procedure SetCursorToEnd(On : Boolean);
```

Purpose

Turn CursorToEnd option on or off.

Comments

Ordinarily, when the cursor is first moved to a field that already contains text, it is positioned at the beginning of the field. SetCursorToEnd allows you to specify that the cursor be positioned at the end of the field instead, so that it is in position for new text to be appended.

Example

```
SetCursorToEnd(On);
AddStringField(ESR, 'Name', 7, 19, '', 7, 25, 30, 1, nil, Scrap.Name);
SetCursorToEnd(Off);
```

If the Name field already contains text, the cursor will be positioned just beyond the end of the string when it is moved to this field.

See Also

SetClearFirstChar

SetDelimiters / SetEntryWindow

Declaration

```
procedure SetDelimiters(var ESR : ESrecord; Left, Right : Char;
                        FieldAttr, SelectAttr : Byte);
```

Purpose

Set field delimiters for the edit screen.

Comments

This option allows you to specify a pair of characters that will be displayed immediately to the left and right of each field in the designated edit screen. The Left and Right parameters indicate which characters are to be used as delimiters. FieldAttr specifies the attribute that the delimiters are to be displayed in when the field they go with is inactive. And SelectAttr specifies the attribute they are to be displayed in when the field they go with is selected for editing. By default, the field delimiters for an edit screen are both nulls (#0), which disables this option.

Example

```
SetDelimiters(ESR, '[', ']', $0F, $70);
```

Requests that all fields be delimited by brackets, and that the brackets be displayed in white on black for inactive fields and reverse video for selected fields.

Declaration

```
procedure SetEntryWindow(var ESR : ESrecord; XLow, YLow, XHigh, YHigh : Byte;
                         Framed : Boolean; BackAttr, FrameAttr : Byte);
```

Purpose

Set coordinates and background color for a data entry window.

Comments

This routine allows you to specify the screen coordinates for the window in-which a data entry screen is displayed. BackAttr specifies the video attribute used when clearing the window (with ClearEntryWindow) and when scrolling the window. If this routine is not called, the entire screen is the window, and the value of TextAttr at the time InitESrecord was called is used as the background attribute.

Framed indicates whether or not the window should have a frame. If it is True, the frame characters currently in FrameChars (TPCRT) will be used to draw the frame. FrameAttr gives the color for the frame. Note too that if Framed is True, the screen coordinates specified determine where the frame is drawn, and the coordinates for the actual data entry window will be adjusted accordingly.

Example

```
SetEntryWindow(ESR, 1, 5, 80, 20, False, TextAttr, TextAttr);
```

This selects all of rows 5-20 as the window in which the data entry screen associated with ESR is to be displayed. The window is unframed.

See Also

ClearEntryWindow InitESrecord

Declaration

```
procedure SetErrorPtr(var ESR : ESrecord; P : Pointer);
```

Purpose

Set pointer to a routine to be called after error.

Comments

This procedure allows you to designate a routine that is to be called each time the edit screen manager encounters an error. If no error handler is installed, the error will be signalled by a beep, and no error message will be displayed. See the section on "Error Handling" at the beginning of this chapter for more details, and ENTRY.PAS for an example of how to use this facility.

Example

```
{$F+}
procedure ErrorHandler(var ESR : ESrecord; Code : Byte; Msg : string);
begin
  {display error message and wait for keypress}
  {clear portion of screen overwritten by message}
end;
{$F-}
...
SetErrorPtr(ESR, @ErrorHandler);
```

In this example, ErrorHandler will be called each time an error occurs. It will display Msg at a designated location on the screen, wait for a keypress, then clear the portion of the screen that was overwritten by the message.

SetExitOnSecondClick / SetFieldAttr

Declaration

```
procedure SetExitOnSecondClick(On : Boolean);
```

Purpose

Allow double-clicking on a field to produce the ESclickExit exit command.

Comments

This routine, which is included only if 'UseMouse' was defined when TPENTRY was compiled, provides a useful hook for programs that include mouse support. Probably the best way to explain what it's good for is to give a concrete example.

In ENTRY.PAS, the <F2> key is assigned to the user-defined exit command, ESuser1. If this command is given at a time when State is the current field, ENTRY displays a pick list of abbreviated state names that the user can choose from. In order to allow the mouse to simulate the pressing of <F2>, the function call that adds the State field is surrounded by calls to SetExitOnSecondClick:

```
SetExitOnSecondClick(On);
AddStringField(ESR, 'State', 10, 18, 'AA', 10, 25, 02, 4, @ValidateNotPartial,
  Scrap.State);
SetExitOnSecondClick(Off);
```

This tells EditScreen to exit with a return code of ESclickExit if the user clicks the left mouse button (i.e. gives the ESmouse command) when 1) State is already the current field and 2) the mouse cursor is on top of it. This gives ENTRY an opportunity to display its pick list.

The name of this routine stems from the fact that the left mouse button will usually have to be clicked twice for a situation like this to occur: once to select the current field, and a second time to generate the exit command.

Declaration

```
procedure SetFieldAttr(A : Byte);
```

Purpose

Set field attribute.

Comments

This routine lets you set the attribute (called the field attribute) used to display fields when they are *not* selected for editing. The value passed to SetFieldAttr is stored in the global variable ESfieldAttr.

Example

```
SetStringAttr($70);
SetFieldAttr($0F);
```

Use reverse video for strings being edited; display inactive fields in white on black.

See Also

SetCtrlAttr
SetPromptAttr
SetStringAttr

Declaration

```
procedure SetFieldLinks(var ESR : ESrecord; FieldID, Next, Prev : Word);
```

Purpose

Specify the fields to jump to when <Enter> or <ShiftTab> is pressed.

Comments

This routine allows you to modify the order in which fields are "visited" as you move through an entry screen by specifying explicit backward and forward links. It can be called at any time after the field associated with FieldID is added to the entry screen, even if the fields denoted by Next and Prev have not been added yet.

ESR is the entry screen, and FieldID is the field whose forward and/or backward links you want to change. Next is the ID for the field to jump to when <Enter> is pressed (or any other event that causes the cursor to be moved to the next field). Prev is the ID for the field to jump to when <ShiftTab> is pressed (or any other event that causes the cursor to be moved to the previous field).

The default value for all forward and backward links is 'BadFieldId' (= $FFFF). When a link is BadFieldId, EditScreen simply applies its usual set of rules for determining where to move the cursor. That is, it moves the cursor to the next or previous unprotected field, wrapping to the top or bottom of the entry screen if necessary and if the current wrap mode allows it.

Although it is not possible to change only one link (backward *or* forward) for a field, specifying BadFieldId for the link that should not change usually achieves the same result. This will not work if the link was previously changed from its default value by a call to SetFieldLinks or SetAllFieldLinks.

Although SetAllFieldLinks is easier to use when numerous changes must be made, SetFieldLinks provides greater flexibility. Multiple fields can safely be given the same forward link, for example, and individual links can be changed easily at any time--even from within a post-edit routine.

Note that calls to SetFieldLinks have no effect on the behavior of the <Up> and <Down> commands. Those commands always move the cursor to the field immediately above or below the current field. If you want <Up> and <Down> to move the cursor to the next/previous field, reassign them to the ESprevField and ESnextField commands, respectively, using the AddEntryCommand routine described in Appendix B.

Example

```
SetFieldLinks(ESR, 0, 1, 2);
SetFieldLinks(ESR, 1, 3, 0);
SetFieldLinks(ESR, 2, 0, 4);
SetFieldLinks(ESR, 3, 4, 1);
SetFieldLinks(ESR, 4, 2, 3);
...
ExitCommand := EditScreen(ESR, 2, False);
```

The entry screen is set up such that the cursor starts at field 2, then moves to fields 0, 1, 3, 4 (and then back to 2) in that order. Contrast this example with the one for SetAllFieldLinks, which does the same thing.

SetForceMode / SetInsertPushes

Declaration

```
procedure SetForceMode(Force, Overtype : Boolean);
```

Purpose

Force insert or overtype mode, else use previous setting.

Comments

Ordinarily, insert mode is on by default, and if the user switches to overtype mode the editors will remain in overtype mode until the user changes modes. In some cases, however, it may be desirable to force insert or overtype mode on temporarily for a given field. SetForceMode allows you to request that a particular mode be forced on each time that field is edited. The user is still allowed to change modes, however. You can control only what the mode is when editing starts.

If the Force parameter is True, then the Overtype parameter indicates whether insert or overtype mode is to be forced on. If the Force parameter is False, the Overtype parameter is ignored, and insert/overtype mode is handled normally.

Example

```
SetForceMode(True, False);
AddRealField(ESR, 'Hourly wage', 16, 12, '$99.99', 16, 25, 12, 0, 999.99, 0,
  Scrap.Wage);
SetForceMode(False, False);
```

Each time the Wage field is edited, the current mode (insert or overtype) will be saved, and insert mode will be forced on. After the field has been edited, the mode will be restored to what it was previously.

Declaration

```
procedure SetInsertPushes(On : Boolean);
```

Purpose

Turn the InsertPushes option on or off.

Comments

By default, any text entered in a subfield when insert mode is on causes all text to the right of the cursor to be pushed to the right, and if there is a character at the end of the subfield it will be pushed off the end and lost. You can change this behavior by executing "SetInsertPushes(Off)". When the InsertPushes option is off, characters cannot be pushed off the end, and text may be inserted only if the last character in the current subfield contains a blank space. Note that this setting affects the behavior of only the string editor. The numeric editor never allows text to be inserted when the current field is full.

Example

```
SetInsertPushes(Off);
AddStringField(ESR, 'City', 9, 19, '', 9, 25, 25, 3, nil, Scrap.City);
SetInsertPushes(On);
```

InsertPushes is off while City is edited, and on while the other fields are edited.

Declaration

```
procedure SetNumeric(On : Boolean);
```

Purpose

Activate or deactivate use of the numeric editor.

Comments

This routine allows you to request that particular fields be edited using the numeric (right-to-left, or calculator-style) editor. Note that this editor should be used only for fields of a numeric type: Byte, Integer, LongInt, Real, Double, etc. See the section on "The Three Field Editors" at the beginning of this chapter for more details, and ENTRY.PAS for an example of how to use this facility.

Example

```
SetNumeric(On);
AddRealField(ESR, 'Hourly wage', 16, 12, '$99.99', 16, 25, 12, 0, 999.99, 0,
  Scrap.Wage);
SetNumeric(Off);
```

In this example, the Wage field will be edited using the numeric editor, while previous and subsequent fields will be edited with the string editor.

See Also

SetAutoNumLock

Declaration

```
procedure SetPadChar(Ch : Char);
```

Purpose

Set the character used to pad the ends of strings.

Comments

This routine allows you to specify the character used to pad strings to fill out a field. The default pad character is a space (' '). Only one end of the string is padded. If the field is left-justified, as most fields are, it will be the right end. If it is right-justified, it will be the left end. See ENTRY.PAS for some examples of the uses of SetPadChar.

Example

```
SetPadChar('*');
AddRealField(ESR, 'Weekly', 16, 49, '$###,###.##', 16, 57, 13, 0, 0, 0,
  Scrap.Weekly);
SetPadChar(' ');
```

This sets the pad character to an asterisk, adds a field, then resets the pad character to a space.

See Also

SetRightJustify

SetPasswordChar / SetPasswordMode

Declaration

```
procedure SetPasswordChar(Ch : Char);
```

Purpose

Set character used in password mode.

Comments

This routine allows you to specify the character used to represent text while in password mode. The default value is ^G (a dot in the center of the character box). Note that this procedure has global effects. You cannot have a different password character for different fields in an entry screen.

Declaration

```
procedure SetPasswordMode(On : Boolean);
```

Purpose

Turn password mode on/off.

Comments

This routine allows you to designate particular fields for editing in "password mode." In this mode, display of any text entered into the field is suppressed, and the character designated by a call to SetPasswordChar will be used to represent all characters in a string. This routine is intended to be used only with string fields.

Example

```
SetPasswordChar('*');
SetPasswordMode(On);
EditString(ESR, 'Enter password: ', 1, 1, 10, '', 0, Escaped, Password);
SetPasswordMode(Off);
```

In this case, all characters entered by the user would be displayed as asterisks.

Declaration

```
procedure SetPostEditPtr(var ESR : ESrecord; P : Pointer);
```

Purpose

Specify pointer to a routine to be called just after a field has been edited.

Comments

This procedure allows you to designate a routine that is to be called just after a field has been edited. The most likely use for this facility is to implement calculated fields. See the section on "Calculated Fields" at the beginning of this chapter for more details, and ENTRY.PAS for an example of how to use this facility.

Example

```
{$F+}
procedure CalculatedFieldHandler(var ESR : ESrecord);
begin
  case ESR.CurrentID of
    {update any fields affected by changes to the current field}
  end;
end;
{$F-}
...
SetPostEditPtr(ESR, @CalculatedFieldHandler);
```

In this example, CalculatedFieldHandler will get called just after a field has been edited, and it can check ESR.CurrentID to determine which of the other fields, if any, need to be updated.

See Also

SetProtection

SetPreEditPtr / SetPromptAttr

Declaration

```
procedure SetPreEditPtr(var ESR : ESrecord; P : Pointer);
```

Purpose

Specify pointer to a routine to be called just before a field is edited.

Comments

This procedure allows you to designate a routine that is to be called just before a new field is edited. The most likely use for this facility is to display a help message associated with the field about to be edited. See the section on "Displaying Help Messages" at the beginning of this chapter for more details, and ENTRY.PAS for an example of how to use this facility.

Example

```
{$F+}
procedure DisplayHelpMessage(var ESR : ESrecord);
begin
  case ESR.CurrentID of
    {display help message for this field}
  end;
end;
{$F-}
...
SetPreEditPtr(ESR, @DisplayHelpMessage);
```

In this example, DisplayHelpMessage will be called before a new field is edited, and it uses ESR.CurrentID to determine which field's help message should be displayed.

Declaration

```
procedure SetPromptAttr(A : Byte);
```

Purpose

Set prompt attribute.

Comments

This routine lets you set the attribute (called the prompt attribute) used to display the prompt associated with a given field. The value passed to SetPromptAttr is stored in the global variable ESpromptAttr.

Example

```
SetStringAttr($70);
SetPromptAttr($07);
```

Display strings being edited in reverse video, prompts in light gray on black.

See Also

SetCtrlAttr
SetFieldAttr
SetStringAttr

Declaration

```
procedure SetProtectAttrs(PromptA, FieldA : Byte);
```

Purpose

Set the prompt and field attributes to be used for protected fields.

Comments

This routine allows you to specify the attributes to be used when displaying protected fields. When using it, you should keep in mind that subsequent calls to SetPromptAttr and/or SetFieldAttr will override the settings specified here. (Those two routines set the default attributes for both protected and unprotected fields.)

Examples

```
SetProtectAttrs(7, 7);
SetPromptAttr($B);
SetFieldAttr($F);
```

Wrong! The settings specified in the call to SetProtectAttrs will be overridden by the subsequent calls to SetPromptAttr and SetFieldAttr.

```
SetPromptAttr($B);
SetFieldAttr($F);
SetProtectAttrs(7, 7);
```

Correct.

Declaration

```
procedure SetProtection(On : Boolean);
```

Purpose

Turn the protected field option on or off.

Comments

This routine allows you to designate particular fields as protected, meaning that they may not be edited. Besides allowing for display-only fields, this option is essential in the creation of calculated fields. For several examples of the uses of protected fields, see the example program ENTRY.PAS.

Example

```
SetProtection(On);
AddRealField(ESR, 'Weekly', 16, 49, '$###,###.##', 16, 57, 13, 0, 0, 0,
  Scrap.Weekly);
SetProtection(Off);
```

In this case, the Weekly field is marked as protected because its value is going to be calculated by the program rather than entered by the user.

See Also

ChangeProtection

SetPostEditPtr

SetRequired / SetRightJustify

Declaration

```
procedure SetRequired(On : Boolean);
```

Purpose

Turn required field option on or off.

Comments

This routine allows you to designate particular fields as required, meaning that they may not be left entirely blank. Note that this is not the same thing as verifying that a field is completely full. If more stringent validation is needed for a given field, you should use a special validation routine such as ValidateNoBlanks.

Example

```
SetRequired(On);
AddStringField(ESR, 'Name', 7, 19, '', 7, 25, 30, 1, nil, Scrap.Name);
SetRequired(Off);
```

The Name field is marked as required so that the user cannot leave it blank.

Declaration

```
procedure SetRightJustify(On : Boolean);
```

Purpose

Turn right justification on/off.

Comments

This routine lets you request that a field be right-justified in its field. It is intended primarily for use with fields containing numbers. Note, however, that numeric fields--those added after a SetNumeric(On) statement--and fields containing fixed decimal points are automatically right-justified. If the field being right-justified is not a number, it is wise to select the blank trimming option for the field as well.

Be aware that, except in the cases of numeric fields and fields with fixed decimal points, right-justification occurs only at the entry and exit points for the field. So, if the string being edited contains '123' to begin with and the picture mask is '99999', it will be displayed as ' 123' initially. If the user then enters '456', '456 ' will be displayed until the field is exited, when it will be redisplayed as ' 456'.

Example

```
SetRightJustify(On);
AddByteField(ESR, 'Age', 14, 52, '999', 14, 57, 10, 0, 0, Age);
SetRightJustify(Off);
```

This turns right-justification on, adds the Age field, then turns right-justification back off so that the next field added will be left-justified.

See Also

SetPadChar

SetTrimBlanks

Declaration

```
procedure SetStringAttr(A : Byte);
```

Purpose

Set string attribute.

Comments

This routine lets you set the attribute (called the string attribute) used to display fields when they are selected for editing. The value passed to SetStringAttr is stored in the global variable ESstringAttr.

Example

```
SetStringAttr($70);
```

Display strings being edited in reverse video.

See Also

SetCtrlAttr
SetFieldAttr
SetPromptAttr

Declaration

```
procedure SetTrimBlanks(On : Boolean);
```

Purpose

Turn blank trimming on or off.

Comments

This routine allows you to determine whether or not leading and trailing spaces are stripped from an input string before the string is stored. Let's say, for example, that the picture mask for a field is '!!!!!', and the user enters ' XYZ '. If blank trimming is on, the result is 'XYZ'. If it is off, the result is whatever was entered.

SetTrimBlanks should be called before adding the first field that it is meant to affect. By default, blank trimming is on. Note that blank trimming is automatically disabled for fields whose picture masks contain literal or semi-literal characters.

Example

```
SetTrimBlanks(Off);
AddStringField(ESR, 'Notes', 19, 18, CharStr('X', 254), 19, 25, 46,16, nil,
  Scrap.Notes);
SetTrimBlanks(On);
```

Since blank trimming is off, Scrap.Notes will always be padded with blanks to its maximum length of 254.

See Also

AddArrayField

SetUpdatePtr

Declaration

```
procedure SetUpdatePtr(var ESR : ESrecord; P : Pointer);
```

Purpose

Set a pointer to a routine to call after the entire entry screen is redrawn.

Comments

This routine allows you to specify the address of a procedure that will be called each time that the specified entry screen (ESR) is redrawn in its entirety. Your procedure should be of the form:

```
{$F+}
procedure MyUpdateProc(var ESR : ESrecord);
begin
end;
{$F-}
```

The primary purpose for this hook is to allow ordinary text strings to be displayed within a scrollable entry screen.

If ESR is indeed scrollable, then its virtual screen will be active at the time your update procedure is called. You may write text to the virtual screen using FastWrite or any of its kin (see TPWINDOW in section 2.C for a discussion of how to write to virtual screens). For example,

```
FastWrite('------ Divider ------', 75, 20, $07);
```

would write the specified text to row 75 of the virtual screen.

If ESR is not scrollable, your update procedure will be writing to the physical screen, rather than a virtual screen. Please note that, although an entry screen has window coordinates associated with it, TPENTRY does not call the Window() procedure in TPCRT, so you should not use routines that take window-relative coordinates (such as FastWriteWindow) to display text inside the entry window. Use routines (such as FastWrite) that take absolute coordinates instead.

Declaration

```
procedure SetWrapMode(var ESR : ESrecord; WrapMode : WrapModeType);
```

Purpose

Select the wrap mode for an edit screen.

Comments

This procedure allows you to control the behavior of the edit screen manager in cases where the cursor is at the last or first row in the edit screen, and the user has given a command that requires it to be moved to the next or previous row, respectively. The default wrap mode is StopAtEdges, which means that any such command is ignored, and the cursor stays in the current field. Probably the most desirable alternative is WrapAtEdges, which causes the cursor to "wrap" at the edges of the edit screen, meaning that the cursor will be moved from the last row to the first or vice versa. The other alternative is ExitAtEdges, which causes any request to go beyond an edge to be interpreted as an exit command. For example, if the cursor is on the top row and the user presses <Up>, EditScreen will exit with a function result of ESprevRec (edit previous record), just as if <PgUp> had been pressed. If the cursor were on the last row and the user pressed <Down>, the result would be ESnextRec (edit next record).

Example

```
SetWrapMode(ESR, WrapAtEdges);
```

Requests that the cursor be moved to the first row when the last row is passed, and vice versa.

See Also

SetAutoAdvance

ValidateNoBlanks / ValidateNotPartial

Declaration

```
function ValidateNoBlanks(var FR : FieldRec;
                          var ErrCode : Byte;
                          var ErrorSt : StringPtr) : Boolean;
```

Purpose

Validate that no usable subfields in a string contain spaces.

Comments

This validation routine verifies that a field is completely full. Like all high-level validation routines, it should never be called directly. It should be referred to only by its address. Note that use of this validation routine imposes more stringent requirements than designating a field as required, since required fields may contain partial entries.

Example

```
AddStringField(ESR, 'State', 10, 18, 'AA', 10, 25, 2, 4, @ValidateNoBlanks,
  Scrap.State);
```

By specifying ValidateNoBlanks as the validation routine for this field, we insure both that the user will not leave the field blank and that he cannot enter a single letter for the state.

See Also

SetRequired

Declaration

```
function ValidateNotPartial(var FR : FieldRec;
                            var ErrCode : Byte;
                            var ErrorSt : StringPtr) : Boolean;
```

Purpose

Validate that no usable subfields in a string contain spaces unless all do.

Comments

This validation routine verifies that a field is either completely full or completely empty. Like all high-level validation routines, it should never be called directly. It should be referred to only by its address.

Example

```
AddStringField(ESR, 'State', 10, 18, 'AA', 10, 25, 2, 4, @ValidateNotPartial,
  Scrap.State);
```

By specifying ValidateNotPartial as the validation routine for this field, we both insure that the user does not enter a single letter for the state and allow him to leave the field blank if the state is unknown.

Declaration

```
function ValidateSubfields(SubfieldMask : string;
                           var FR : FieldRec;
                           var ErrCode : Byte;
                           var ErrorSt : StringPtr) : Boolean;
```

Purpose

Validate that subfields in a string meet the requirements of SubfieldMask.

Comments

This is a low-level routine that should be called only by a higher-level validation routine. It is intended to assist the programmer in writing custom validation routines for fields containing multiple subfields, some of which are required and some of which are not.

A SubfieldMask must be of the same length as the picture mask for the field being validated. The following characters are meaningful in a SubfieldMask:

```
r (ReqdChar)    - subfield is Required, no blank spaces allowed
u (UnlessChar)  - subfield is required, Unless entire field is empty
p (PartialChar) - subfield is optional, but cannot be Partially full
```

Note that these characters may not be mixed within a given subfield.

See the demonstration program ENTRY.PAS for two typical examples of the use of ValidateSubfields.

Example

```
const
  PhoneMask   : string[14] = '(999) 999-9999';
  ValidPhone  : string[14] = '(ppp) uuu-uuuu';
...
{$F+}
function ValidatePhone(var FR : FieldRec; var ErrCode : Byte;
                       var ErrorSt : StringPtr) : Boolean;
begin
  ValidatePhone := ValidateSubfields(ValidPhone, FR, ErrCode, ErrorSt);
end;
{$F-}
```

Here ValidatePhone is a high-level validation routine which uses ValidateSubfields to verify that the last two subfields are not empty unless the entire field is empty, and that the first subfield is either completely full or completely empty.

See Also

ValidateNoBlanks

ValidateNotPartial

C. TPDATE: Date and Time Variables

TPDATE specializes in history. Its primary function is to provide routines that allow you to store dates and times in compact formats, to convert them to other forms when necessary, and to perform date/time arithmetic.

One of its chief strengths is that TPDATE provides numerous functions for converting Julian dates and times to strings and back again. To allow the maximum amount of flexibility, and to insure that the same routines can be used internationally, TPDATE lets you specify a "picture mask" to indicate how a given string should be formatted. For example, let's say that a Julian date variable contains January 2, 1900 and you want to convert it to a string (using the DateToDateString routine). There are several possible picture masks you could use:

Picture	Result
'mm/dd/yy'	'01/02/00'
'MM/dd/yy'	' 1/02/00'
'mm/dd/yyyy'	'01/02/1900'
'mm-dd-yyyy'	'01-02-1900'
'dd/mm/yyyy'	'02-01-1900'
'DD/mm/yyyy'	' 2-01-1900'

Time strings are handled in similar fashion, except that special formatting characters are available for working with times in am/pm (12-hour) format as well as military (24-hour) format. For example, let's say that you wanted to convert a Time variable containing 1:02 am into a string (using TimeToTimeString). Here are some possible picture masks:

Picture	Result
'hh:mm:ss'	'01:02:00'
'hh:mm'	'01:02'
'hh:mm te'	'01:02 am'
'hh:mmt'	'01:02a'

Note that string representations of time are always assumed to be in military format unless a 't' appears in the picture mask. An 'e' (em) in the mask should always be accompanied by a corresponding 't' (time).

Now that you have a general idea of what picture masks do, let's look at the characters that can appear in them. The following special characters are meaningful in picture masks:

Char	Marks location of or is replaced by
'm', 'M'	month (date fields only)
'd', 'D'	day (date)
'y'	year (date)
'/'	'/' or SlashChar (date; semi-literal)
'h', 'H'	hours (time fields only)
'm', 'M'	minutes (time)
's', 'S'	seconds (time)
':'	':' or ColonChar (date; semi-literal)
't'	'p', 'P', 'a', 'A' (time; case depends on UpCaseTime)
'e'	'm' or 'M' (time; case depends on UpCaseTime)

The difference between the upper- and lower-case formatting characters is that the lower-case ones cause numbers to be padded with '0', while the upper-case ones cause them to be padded with spaces. For example, if the date is 1/1/1900, a picture mask of 'MM/dd/yyyy' would produce ' 1/01/1900', while a mask of 'mm/dd/yyyy' would produce '01/01/1900'. 'Y' is not allowed as a formatting character because 1) it would probably never be used, and 2) it would create a conflict with one of the formatting characters in TPENTRY, which depends on TPDATE.

Note that two of these characters, '/' and ':', are considered "semi-literal". What this means is that when TPDATE sees one of these two characters in a picture mask, it automatically replaces them with the current value of SlashChar (for '/') or ColonChar (for ':'). This allows you, for example, to write a program that contains numerous picture masks like 'mm/dd/yy', for example, and have all of them produce strings in 'mm-dd-yy' format simply by executing the statement "SlashChar := '-';". Alternatively, you could put the '-' in the mask to begin with: 'mm-dd-yy'. Given the correct picture mask and the right settings for ColonChar and SlashChar, TPDATE is thus able to deal with almost any country's conventions for displaying dates and times. (See the documentation for the InternationalDate and InternationalTime routines, as well as the section on "International Currencies and Conventions" earlier in the chapter.)

Any characters not shown in the list above are treated as literals when they appear in a picture mask.

TPDATE does much more than convert dates and times to strings, however. There are several other noteworthy features of the unit, such as the routines for performing date and time arithmetic, as you'll see when you study the remainder of this chapter.

Valid Range for Dates

TPDATE is designed to allow dates to be stored as either long integers (giving a range of 1/1/1600-12/31/3999) or words (giving a range of 1/1/1900-12/31/2078). The factor determining which method of storage is used is the conditional compilation

directive FourByteDates. By default, FourByteDates is defined, and long integers are used for storage. If you would prefer that dates be stored as words to conserve space, all you need to do is delete the statement '{$DEFINE FourByteDates}' in TPDEFINE.INC and recompile TPDATE. Among other things, this change will alter the values of several important constants interfaced by the unit and described later in this chapter: MinYear, MaxYear, MinDate, MaxDate, Date1900, and BadDate. Note that it is assumed throughout the rest of the manual that FourByteDates is defined.

About the Term "Julian Date"

The term "Julian date" means different things to different people. Among astronomers and historians, it has a very precise and technical meaning, and our use of the term will probably raise some eyebrows, because TPDATE doesn't work with "true" Julian dates. Among programmers, however, the phrase has acquired a much broader meaning, something like "a date stored as an integer value representing the serial number of days from a given point in time." And it is in that (widely understood) sense that we use the term here.

Types

```
{$IFDEF FourByteDates}
Date = LongInt;
{$ELSE}
Date = Word;
{$ENDIF}
```

In TPDATE, dates are stored in Julian format as the number of days since January 1, 1600. Besides offering compact storage--any date through December 31, 3999 can be stored in four bytes--this method facilitates date arithmetic, since the number of days between any two dates can be determined with simple subtraction. Note, however, that if the conditional compilation directive FourByteDates is not defined, dates are stored as words (two bytes), giving a range of 1/1/1900 to 12/31/2078.

```
DateString = string[DateLen];
```

A string representation of a date.

```
DateTimeRec =
  record
    D : Date;
    T : Time;
  end;
```

This record type simply combines the two basic date types defined by TPDATE, Date and Time.

```
DayType = (Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday);
```

An enumerated type used when representing a day of the week.

```
Time = LongInt;
```

TPDATE handles time in a manner similar to dates, representing a given time of day as the number of seconds since midnight. The valid range for a time variable is thus 0 (12 am) to 86,399 (11:59:59 pm), since there are 86,400 (60*60*24) seconds in a day. This method is reasonably compact and facilitates time arithmetic.

Constants

```
{$IFDEF FourByteDates}
BadDate = $FFFFFFFF;
{$ELSE}
BadDate = $FFFF;
{$ENDIF}
```

This value is used to represent an invalid date, such as 1/32/1988.

```
BadTime = $FFFFFFFF;
```

This value is used to represent an invalid time of day, such as 12:61:00.

```
ColonChar : Char = ':';
```

This typed constant determines what character will be used as a separator in time strings. For example, if TimeToTimeString is passed a picture mask of 'hh:mm:ss' and ColonChar is '.', the string returned will be in 'hh.mm.ss' format.

```
{$IFDEF FourByteDates}
Date1900 = $0001AC05;
{$ELSE}
Date1900 = $0000;
{$ENDIF}
```

This constant contains the Julian date for 1/1/1900.

```
{$IFDEF FourByteDates}
Date1980 = $00021E28;
{$ELSE}
Date1980 = $7223;
{$ENDIF}
```

This constant contains the Julian date for 1/1/1980.

```
{$IFDEF FourByteDates}
Date2000 = $00023AB1;
{$ELSE}
Date2000 = $8EAC;
{$ENDIF}
```

This constant contains the Julian date for 1/1/2000.

```
DateLen = 20;
```

This is the maximum length of a Picture string passed to a routine in TPDATE.

```
DayString : array[DayType] of string[9] = (
  'Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday',
  'Saturday');
```

This array can be used to convert DayType variables to strings. For example, if DayOfWeek = Saturday, then the statement 'S := DayString[DayOfWeek]' would set S to 'Saturday'. Notice that you could then get a reasonable abbreviation for the day of the week using 'Copy(S, 1, 3)'.

```
HourOnly    = 'h';
MinOnly     = 'm';
SecOnly     = 's';
HourOnlyU   = 'H';
MinOnlyU    = 'M';
SecOnlyU    = 'S';
TimeOnly    = 't';
EmOnly      = 'e';
TimeColon   = ':';
```

These formatting characters are the only ones that are meaningful in a picture mask for a time string. The uppercase versions of the formatting characters cause numbers to be left-padded with spaces (' 1:11:00'), while the lowercase versions cause them to be padded with 0 ('01:11:00'). See the introduction to this chapter for more details.

```
{$IFDEF FourByteDates}
MinDate = $00000000; {= 01/01/1600}
MaxDate = $000D6025; {= 12/31/3999}
{$ELSE}
MinDate = $0000;     {= 01/01/1900}
MaxDate = $FF62;     {= 12/31/2078}
{$ENDIF}
```

These constants define the valid range of dates for a date variable.

```
MinTime = 0;     {= 00:00:00 am}
MaxTime = 86399; {= 23:59:59 pm}
```

These constants define the valid range of times for a time variable.

```
{$IFDEF FourByteDates}
MinYear = 1600;
MaxYear = 3999;
{$ELSE}
MinYear = 1900;
MaxYear = 2078;
{$ENDIF}
```

These constants define the valid range of years for a date variable.

```
MonthOnly   = 'm';
DayOnly     = 'd';
YearOnly    = 'y';
MonthOnlyU  = 'M';
DayOnlyU    = 'D';
DateSlash   = '/';
```

These formatting characters are the only ones that are meaningful in a picture mask for a date string. The uppercase versions of the formatting characters cause numbers to be left-padded with spaces (' 1/11/1900'), while the lowercase versions cause them to be padded with '0' ('01/11/1900'). See the introduction to this chapter for more details.

```
MonthString : array[1..12] of string[9] = (
  'January', 'February', 'March', 'April', 'May', 'June', 'July',
  'August', 'September', 'October', 'November', 'December');
```

This array can be used to convert variables representing months to strings. For example, if Month = 1, then the statement 'S := MonthString[Month]' would set S to 'January'. Notice that you could then get a reasonable abbreviation for the month using 'Copy(S, 1, 3)'.

```
HoursInDay      = 24;
MinutesInHour   = 60;
SecondsInDay    = 86400;
SecondsInHour   = 3600;
SecondsInMinute = 60;
```

These self-explanatory constants are interfaced to facilitate time arithmetic.

```
SlashChar : Char = '/';
```

This typed constant determines what character will be used as a separator in date strings. For example, if DateToDateString is passed a picture mask of 'mm/dd/yy'

and SlashChar is '-', the string returned will be in 'mm-dd-yy' format.

```
Threshold2000 : Integer = 1900;
```

This constant determines how TPDATE interprets the year portion of a date like 01/01/40. If Threshold2000 is in the range 1900..1940, inclusive, the year is assumed to be 1940. If Threshold2000 is in the range 1941..2000, the year is assumed to be 2040. More generally, Threshold2000 specifies the year at which TPDATE stops assuming a century of 2000 when the year is in the range 00..99 and starts assuming 1900. With the default value of 1900, 1900 is always assumed. With a value of 2000, 2000 would always be assumed.

```
UpCaseTime : Boolean = False; {if true, 't' and 'e' force upper case}
```

This value affects the behavior of routines that return string representations of times. If it is true, and the picture mask passed to the routine contains 't' and/or 'e', the string returned will contain 'P', 'PM', 'A', or 'AM' rather than their lower-case equivalents.

Variables

```
DefaultMonth : Byte;
DefaultYear : Integer;
```

These variables are initialized to the current month and year. They are used by DateStringToDMY and DateStringToDate when a partial date is given. For example, if the date string contains '12/01', the picture mask is 'mm/dd', and DefaultYear is 1989, then the date is interpreted as 12/01/1989.

Declaration

```
function CurrentTime : Time;
```

Purpose

Returns current time in seconds since midnight.

Comments

This function calls DOS to determine the current time of day, then converts the values it returns (hours, minutes, seconds) to a time variable, which is returned as the function result.

Example

```
var Start, Elapsed : Time;
...
Start := CurrentTime;
...
Elapsed := CurrentTime-Start;
if Elapsed < 0 then
  Inc(Elapsed, SecondsInDay);
```

Here CurrentTime is used to calculate the number of seconds that have elapsed between two events.

Declaration

```
function CurrentTimeString(Picture : DateString) : DateString;
```

Purpose

Returns current time as a string of the specified form.

Comments

This function simply calls CurrentTime to get the time of day, then converts it to a string of the form specified by Picture.

Example

```
WriteLn('The time is ', CurrentTimeString('hh:mm:ss te'));
```

Display the current time of day.

DateDiff / DateStringToDate

Declaration

```
procedure DateDiff(Date1, Date2 : Date; var Days, Months, Years : Integer);
```

Purpose

Return the difference in Days, Months, Years between two valid Julian dates.

Comments

Given two valid Julian dates, Date1 and Date2, this procedure calculates the absolute difference between them in Days, Months, and Years. The order of Date1 and Date2 is not important.

Example

```
var Days, Months, Years : Integer;
...
DateDiff(Today, Today+1, Days, Months, Years);
```

In this case, Days would be 1, while Months and Years would be 0.

See Also

DateTimeDiff TimeDiff

Declaration

```
function DateStringToDate(Picture, S : DateString) : Date;
```

Purpose

Convert the date string S to a Julian date.

Comments

Given a string representation of a date (S), this routine converts it to a Julian date, returned as the function result. S must be a string of the same length and form as Picture. If the date represented by S is invalid, BadDate is returned.

Example

```
MyDateString := '01/01/1900';
MyDate := DateStringToDate('mm/dd/yyyy', MyDateString);
```

In this case, MyDate would equal the predefined constant Date1900.

See Also

DateStringToDMY DateToDateString

Declaration

```
function DateStringToDMY(Picture, S : DateString;
                         var D, M, Y : Integer) : Boolean;
```

Purpose

Extract day, month, and year from S, returning True if string is valid.

Comments

Given a string representation of a date (S), this routine extracts the day, month, and year, and returns True if the date is valid. S must be a string of the same length and form as Picture.

Example

```
MyDateString := '01/01/1900';
DateOK := DateStringToDate('mm/dd/yyyy', MyDateString, D, M, Y);
```

In this case, D would be 1, M would be 1, Y would be 1900, and DateOK would be True.

See Also

DateStringToDate
DMYtoDateString
DateToDateString

DateTimeDiff

Declaration

```
procedure DateTimeDiff(DT1, DT2 : DateTimeRec;
                       var Days : Word;
                       var Secs : LongInt);
```

Purpose

Return the difference in days and seconds between two points in time.

Comments

Given two variables of type DateTimeRec, DT1 and DT2, this routine returns the absolute difference between the two points in time in terms of Days and seconds (Secs). The order of DT1 and DT2 is not important.

Examples

```
var
  DT1, DT2 : DateTimeRec;
  Days : Word;
  Secs : LongInt;
...
DT1.D := Today;   DT1.T := CurrentTime;
DT2.D := DT1.D+1; DT2.T := DT1.T+1;
DateTimeDiff(DT1, DT2, Days, Secs);
```

In this case, Days and Secs would both be 1.

```
DT2.T := DT1.T-1;
DateTimeDiff(DT1, DT2, Days, Secs);
```

Days would now be 0 and Secs would be 86,399 (SecondsInDay-1).

See Also

DateDiff

TimeDiff

DateToDateString / DateToDMY

Declaration

```
function DateToDateString(Picture : DateString; Julian : Date) : DateString;
```

Purpose

Converts a Julian to a string of the form indicated by Picture.

Comments

Given a date (Julian), this routine converts it to a string of the form specified by Picture and returns it as its function result. Note that if Julian = BadDate, the month, day, and year will be represented as blank spaces.

Example

```
MyDateString := DateToDateString('mm/dd/yy', MyDateVar);
```

Converts MyDateVar to a string of the form 'mm/dd/yy'.

Declaration

```
procedure DateToDMY(Julian : Date; var Day, Month, Year : Integer);
```

Purpose

Convert from a Julian date to day, month, year.

Comments

Given a date (Julian), this routine returns the Day, Month, and Year. If Julian = BadDate, Day, Month and Year are set to 0.

Example

```
var Day, Month, Year : Integer;
...
DateToDMY(Today, Day, Month, Year);
```

Sets Day, Month, and Year to today's date.

See Also

DMYtoDate

DateToSortString / DayOfWeek

Declaration

```
function DateToSortString(Julian: Date) : string;
```

Purpose

Convert a date to a sortable string.

Comments

Given a Julian date, this routine converts it to a string that can be used by sorting and indexing routines that work only with strings, such as those in the Turbo Database Toolbox. The string produced is 4 bytes long and is not in a printable form.

Example

```
var MySortString : string[4];
...
MySortString := DateToSortString(MyDate);
```

This would convert MyDate to a sortable string.

See Also

SortStringToDate

Declaration

```
function DayOfWeek(Julian : Date) : DayType;
```

Purpose

Return the day of the week for the specified date.

Comments

Given a Julian date, this routine returns the corresponding day of the week. Julian is assumed to be a valid date.

Example

```
WriteLn('Today is a ', DayString[DayOfWeek(Today)]);
```

Display the day of the week.

Declaration

```
function DMYtoDate(Day, Month, Year : Integer) : Date;
```

Purpose

Convert from Day, Month, Year to a Julian date.

Comments

Given a date in the form of a Day, Month, and Year, this routine returns the corresponding Julian date. If the date is invalid, BadDate is returned. If Year is less than 100, it is assumed to be relative to 1900.

Example

```
DateToRemember := DMYtoDate(7, 12, 1941);
```

DateToRemember now holds the number of days between January 1, 1600 and December 7, 1941.

See Also

DateToDMY

Declaration

```
function DMYtoDateString(Picture : DateString;
                         Day, Month, Year : Integer) : DateString;
```

Purpose

Merge the day, month, and year into the picture.

Comments

Given a Day, Month and Year, this routine returns a string representation of the date in the format indicated by Picture. If Year is less than 100, it is assumed to be relative to 1900.

Example

```
MyDateString := DMYtoDateString('mm/dd/yyyy', 7, 12, 1941);
```

Sets MyDateString to '12/07/1941'.

See Also

DateStringToDMY

HMStoTime / IncDate

Declaration

```
function HMStoTime(Hours, Minutes, Seconds : Byte) : Time;
```

Purpose

Convert Hours, Minutes, Seconds to a Time variable.

Comments

Given a time of day in terms of Hours, Minutes, and Seconds, this routine returns the time in seconds since midnight. Hours is assumed to be in military format (0-23). If the time is invalid, BadTime is returned.

Example

```
MyTimeVar := HMStoTime(1, 2, 3);
```

MyTimeVar would equal 3723--(1*60*60)+(2*60)+3--or 01:02:03 am.

See Also

TimeToHMS

Declaration

```
function IncDate(Julian : Date; Days, Months, Years : Integer) : Date;
```

Purpose

Add (or subtract) the specified number of Days, Months, and Years to a date.

Comments

Given a valid Julian date, this routine adjusts it by the specified number of Days, Months, and Years. Months and Years are added before days, and no overflow/underflow checks are made because the result is assumed to be a date in the range MinDate..MaxDate.

Examples

```
YearFromNow := IncDate(Today, 0, 0, 1);
```

YearFromNow would hold the Julian representation of the date 1 year from today.

```
YearAgo := IncDate(Today, 0, 0, -1);
```

YearAgo would hold the Julian representation of the date 1 year ago.

Declaration

```
procedure IncDateTime(var DT1, DT2 : DateTimeRec;
                      Days : Integer; Secs : LongInt);
```

Purpose

Increment (or decrement) a date and time by the specified number of days and seconds.

Comments

Given a valid DateTimeRec, DT1, this routine adjusts its date and time by the specified number of Days and seconds, then stores the result in DT2. (DT1 is unchanged.)

Examples

```
var
  StartTime, StopTime : DateTimeRec;
...
IncDateTime(StartTime, StopTime, 0, 8*SecondsInHour);
```

StopTime will have the date and time 8 hours from StartTime.

```
IncDateTime(StartTime, StartTime, -7, 0);
```

StartTime will have the date and time one week ago.

See Also

IncTime

Declaration

```
function IncTime(T : Time; Hours, Minutes, Seconds : Byte) : Time;
```

Purpose

Add the specified hours, minutes, and seconds to T and return the result.

Comments

Given a time of day (T), this routine adds the specified number of Hours, Minutes, and Seconds to it. The result is adjusted if necessary to account for the rollover at midnight.

Examples

```
RightNow := HMStoTime(22, 0, 0); {10:00:00 pm}
HourFromNow := IncTime(RightNow, 1, 0, 0);
```

HourFromNow would equal 11:00:00 pm.

```
ThreeHoursFromNow := IncTime(3, 0, 0);
```

ThreeHoursFromNow would equal 1:00:00 am. (Note the rollover.)

InternationalDate

Declaration

```
function InternationalDate(WholeYear, ZeroPad : Boolean) : DateString;
```

Purpose

Return a picture mask for a date string, based on DOS's country info.

Comments

This routine uses DOS's Get Country Information service in order to determine not only the proper value for SlashChar, but also how to produce a picture mask that is appropriate for the current country.

The WholeYear parameter indicates whether the picture mask should leave room for a century ('yyyy') as well as a year ('yy'). ZeroPad indicates whether the first subfield in the picture mask should be padded with zeroes or with spaces.

Note that the information provided by DOS 2.x is minimal compared to that available under DOS 3.x and higher. We therefore recommend that this function be used with caution in programs that must run under DOS 2.

Examples

```
InternationalDate(True, True)
```

Returns 'mm/dd/yyyy' for US (SlashChar = '-'), 'yyyy/mm/dd' for Sweden (SlashChar = '-').

```
InternationalDate(False, False)
```

Returns 'MM/dd/yy' for US, 'DD/mm/yy' for Switzerland (SlashChar = '.').

See Also

InternationalCurrency (TPENTRY) InternationalTime

InternationalTime / RoundToNearestHour

Declaration

```
function InternationalTime(WithSeconds, ZeroPad : Boolean;
                           ExtraSpace, WithEm : Boolean) : DateString;
```

Purpose

Return a picture mask for a time string, based on DOS's country info.

Comments

This routine uses DOS's Get Country Information service in order to determine not only the proper value for ColonChar, but also how to produce a picture mask that is appropriate for the current country.

WithSeconds indicates whether the mask should make room for seconds as well as hours and minutes. ZeroPad indicates whether the hours subfield should be padded with zeroes or spaces. WithEm indicates whether or not 'te' should be added into the end of the string if appropriate. And, if 'te' is appended, ExtraSpace indicates whether or not there should be a space in front of it.

Note that DOS 2.x provides no information about how time strings should be formatted. (DOS versions numbered 3.0 and higher do.) We therefore recommend that this function not be used in programs that must run under DOS 2.

Examples

```
InternationalTime(True, True, True, True)
```

Returns 'HH:mm:ss te' for US (ColonChar = ':'), 'HH:mm:ss' for Norway (ColonChar = '.').

```
InternationalTime(False, False, False, False)
```

Returns 'hh:mm' for US, 'hh:mm' for Denmark (ColonChar = '.').

See Also

InternationalCurrency (TPENTRY) InternationalDate

Declaration

```
function RoundToNearestHour(T : Time; Truncate : Boolean) : Time;
```

Purpose

Round T to the nearest hour, or truncate minutes and seconds from T.

Comments

This routine does one of two things, depending on the value of Truncate. If Truncate is True, it simply sets the minutes and seconds in T to 0 and returns the value (e.g., 4:01:02 becomes 4:00:00). If Truncate is False, it rounds the time to the nearest hour (e.g., 4:30:00 becomes 5:00:00). T is assumed to be a valid Time variable.

See Also

RoundToNearestMinute

RoundToNearestMinute / SortStringToDate

Declaration

```
function RoundToNearestMinute(T : Time; Truncate : Boolean) : Time;
```

Purpose

Round T to the nearest minute, or truncate seconds from T.

Comments

This routine does one of two things, depending on the value of Truncate. If Truncate is True, it simply sets the seconds in T to 0 and returns the value (e.g., 4:01:02 becomes 4:01:00). If Truncate is False, it rounds the time to the nearest minute (e.g., 4:00:30 becomes 4:01:00). T is assumed to be a valid Time variable.

See Also

RoundToNearestHour

Declaration

```
function SortStringToDate(S : string) : Date;
```

Purpose

Convert a sortable string form to a date.

Comments

Given a sortable string (S) produced by DateToSortString, this routine converts it back to a Julian date, which is returned as the function result.

Example

```
var MySortString : string[4];
    MyDate : Date;
...
MySortString := DateToSortString(MyDate);
...
MyDate := SortStringToDate(MySortString);
```

This would convert MyDate to a sortable string and then back again.

See Also

DateToSortString

Declaration

```
function SortStringToTime(S : string) : Time;
```

Purpose

Convert a sortable string to a time variable.

Comments

Given a sortable string (S) produced by TimeToSortString, this routine converts it back to a time variable, which is returned as the function result.

Example

```
var MySortString : string[4];
    MyTime : Time;
...
MySortString := TimeToSortString(MyTime);
...
MyTime := SortStringToTime(MySortString);
```

This would convert MyTime to a sortable string and then back again.

See Also

TimeToSortString

Declaration

```
procedure TimeDiff(Time1, Time2 : Time; var Hours, Minutes, Seconds : Byte);
```

Purpose

Return the difference in hours, minutes, seconds between two times.

Comments

Given two valid time variables, Time1 and Time2, this procedure calculates the absolute difference between them in Hours, Minutes, and Seconds. The order of Time1 and Time2 is not important.

Example

```
var
  Time1, Time2 : Time;
  Hours, Minutes, Seconds : Byte;
...
Time1 := CurrentTime;
Time2 := IncTime(Time1, 1, 2, 3);
TimeDiff(Time1, Time2, Hours, Minutes, Seconds);
```

In this case, Hours would be 1, Minutes would be 2, and Seconds would be 3.

See Also

DateDiff

DateTimeDiff

TimeStringToTime / TimeToAmPmString

Declaration

```
function TimeStringToTime(Picture, S : DateString) : Time;
```

Purpose

Convert S to a time variable.

Comments

Given a string representation of a time (S), this routine converts it to a time variable, returned as the function result. S must be a string of the same length and form as Picture. If the time represented by S is invalid, BadTime is returned.

Example

```
MyTimeVar := TimeStringToTime('hh:mm:ss', '01:02:03');
```

MyTimeVar would be set to 1:02:03 am.

See Also

TimeToAmPmString TimeToTimeString

Declaration

```
function TimeToAmPmString(Picture : DateString; T : Time) : DateString;
```

Purpose

Convert T to a string of the form indicated by Picture. Times are always displayed in am/pm format.

Comments

This routine is identical to TimeToTimeString except that it forces the time designated by T to be represented in am/pm format, even if the Picture mask does not require it. Note that if Time = BadTime, the hour, minutes, and seconds will be represented as blank spaces.

Example

```
MyTimeString1 := TimeToAmPmString('hh:mm:ss', 0);
MyTimeString2 := TimeToTimeString('hh:mm:ss', 0);
```

MyTimeString1 would be set to '12:00:00', MyTimeString2 to '00:00:00'.

See Also

TimeToTimeString

Declaration

```
procedure TimeToHMS(T : Time; var Hours, Minutes, Seconds : Byte);
```

Purpose

Convert a Time variable to Hours, Minutes, Seconds.

Comments

Given a time variable (T), this routine extracts the Hours, Minutes, and Seconds. If Time = BadTime, Hours, Minutes, and Seconds are all set to 0.

Example

```
var Hours, Minutes, Seconds : Byte;
...
TimeToHMS(CurrentTime, Hours, Minutes, Seconds);
```

Sets Hours, Minutes, and Seconds to the current time.

See Also

HMStoTime

Declaration

```
function TimeToSortString(T : Time) : string;
```

Purpose

Convert a time variable to a sortable string.

Comments

Given a time variable (T), this routine converts it to a string that can be used by sorting and indexing routines that work only with strings, such as those in the Turbo Database Toolbox. The string produced is 4 bytes long and is not in a printable form.

Example

```
var MySortString : string[4];
...
MySortString := TimeToSortString(MyTime);
```

This would convert MyTime to a sortable string.

See Also

SortStringToTime

TimeToTimeString / Today

Declaration

```
function TimeToTimeString(Picture : DateString; T : Time) : DateString;
```

Purpose

Convert T to a string of the form indicated by Picture.

Comments

Given a time of day (T), this routine converts it to a string of the form specified by Picture and returns it as its function result. Note that if Time = BadTime, the hour, minutes, and seconds will be represented as blank spaces.

Example

```
MyTimeString := TimeToTimeString('hh:mm:ss', MyTimeVar);
```

Converts MyTimeVar to a string of the form 'hh:mm:ss'.

Declaration

```
function Today : Date;
```

Purpose

Returns today's date as a julian.

Comments

This function calls DOS to determine the current date, then converts the values it returns (day, month, year) to a date variable, which is returned as the function result.

Example

```
WriteLn('Today is a ', DayString[DayOfWeek(Today)]);
```

Display the day of the week.

See Also

DayOfWeek

TodayString

Declaration

```
function TodayString(Picture : DateString) : DateString;
```

Purpose

Returns today's date as a string of the specified form.

Comments

This function simply calls Today to get the current date, then converts it to a string of the form specified by Picture.

Example

```
WriteLn('Today is ', TodayString('mm/dd/yyyy'));
```

Display today's date.

See Also

Today

Declaration

```
function ValidDate(Day, Month, Year : Integer) : Boolean;
```

Purpose

Verify that Day, Month, Year is a valid date.

Comments

This routine is intended primarily for internal use, but you may use it if you wish. Given a Day, Month, and Year, it returns True if the date is valid and False if it isn't. If Year is less than 100, it is assumed to be relative to 1900. If Year is not less than 100, it must be in the range 1600-3999.

Examples

```
DateIsOK := ValidDate(1, 1, 1598);
```

DateIsOK would be False because the year is out of range.

```
DateIsOK := ValidDate(4, 31, 1988);
```

DateIsOK would be False because there are only 30 days in April.

D. TPMEMO: Memo Field Editor

TPMEMO is a unit containing a simple text editor that can be used in programs written with Turbo Professional. We envision two primary uses for TPMEMO:

First, it can be used in conjunction with TPENTRY for editing "memo fields" in database records. For a demonstration of how to use TPMEMO for this purpose, see ENTRY.PAS, discussed later in "Example Programs".

Second, TPMEMO can be used for editing text files. In this respect and others, TPMEMO is quite similar to Borland's Binary Editor (BINED), part of the Turbo Pascal Editor Toolbox. But there is one important feature in TPMEMO that is lacking in the Binary Editor: word wrap. In fact, one of our main goals in writing TPMEMO was to provide an alternative to BINED in cases where word wrap is essential. For an example of how to use TPMEMO for this purpose, see MEMO.PAS, discussed later in "Example Programs". The two other noteworthy advantages of TPMEMO over BINED are 1) built-in mouse support and 2) complete source code. The one disadvantage of TPMEMO is that it does not implement any of the block manipulation commands found in BINED.

Terminology

Before going any further, a few terms should be defined:

An "edit window" is a region of the screen used to display text being edited with TPMEMO.

An "edit buffer" is a data structure, usually an array of char, used to store the text being edited. The end of an edit buffer is signalled by a ^Z (end of file) character. The end of each line of text within the edit buffer is signalled by a carriage return-line feed combination (^M^J). An empty edit buffer should always be marked by a lone ^Z at the beginning of the buffer.

An "editor control block" is a record variable used to store information about an edit window and the edit buffer associated with it, as well as a variety of status and housekeeping variables. See the entry for EMcontrolBlock in the TPMEMO:Declarations for details on what an editor control block contains.

Editing Commands

TPMEMO offers a fairly complete set of basic editing commands. The available commands are listed below. In each case the first line gives the name of the command, in TPMEMO's terms, followed by the key(s) to which it is normally assigned. The second and following lines give a brief description of the command.

EMleft **<Left>, <CtrlS>**
Cursor left one character.

EMright **<Right>, <CtrlD>**
Cursor right one character.

EMwordLeft **<CtrlLeft>, <CtrlA>**
Cursor left one word. If the cursor is at the beginning of a line, it is moved to the end of the previous line.

EMwordRight **<CtrlRight>, <CtrlF>**
Cursor right one word. If the cursor is at the end of a line, it is moved to the beginning of the following line.

EMhome **<Home>, <CtrlQ><S>**
Cursor to the beginning of the line.

EMend **<End>, <CtrlQ><D>**
Cursor to the end of the line.

EMup **<Up>, <CtrlE>**
Cursor up one line.

EMdown **<Down>, <CtrlX>**
Cursor down one line.

EMscrollUp **<CtrlW>**
Scroll the display up one line.

EMscrollDown **<CtrlZ>**
Scroll the display down one line.

EMpageUp **<PgUp>, <CtrlR>**
Scroll the display up one page.

EMpageDown **<PgDn>, <CtrlC>**
Scroll the display down one page.

EMscreenTop **<CtrlHome>, <CtrlQ><E>**
Move the cursor to the top of the edit window.

EMscreenBot **<CtrlEnd>, <CtrlQ><X>**
Move the cursor to the bottom of the edit window.

EMtopOfFile **<CtrlPgUp>, <CtrlQ><R>**
Move the cursor to the beginning of the file.

EMendOfFile **<CtrlPgDn>, <CtrlQ><C>**
Move the cursor to the end of the file.

EMdel **<Del>, <CtrlG>**
Delete the character at the cursor.

EMback **<Bksp>, <CtrlH>, <CtrlBksp>**
Delete the character to the left of the cursor. If the cursor is at the beginning of a line, the line will be joined with the previous line.

EMdelLine **<CtrlY>**
Delete the current line.

EMdelEol **<CtrlQ><Y>**
Delete from the cursor to the end of the line.

EMdelWord **<CtrlT>**
Delete the word to the right of the cursor. If the cursor is at end of a line, the following line is joined with the current line.

EMenter **<Enter>, <CtrlM>**
Start a new line.

EMtab **<Tab>, <CtrlI>**
Move the cursor to the next tab stop. If insert mode is on, any text to the right of the cursor is moved to the right of the tab stop.

EMctrlChar **<CtrlP>**
Insert a control character. For example, to insert a ^G, you would enter <CtrlP><CtrlG>. This command is available only if the video attribute for control characters is different than that for normal characters. Control characters are displayed as uppercase alphabetics (^G is displayed as 'G', for example) in the specified attribute.

EMins **<Ins>, <CtrlV>**
Toggle insert mode on or off. A fat cursor indicates insert mode; a thin cursor indicates overtype mode.

EMindent **<CtrlO><I>**
Toggle auto-indent mode on or off. In auto-indent mode, pressing <Enter>

while in insert mode causes the new line inserted to have the same indentation level as the previous line. Auto-indent mode also affects the way that text is formatted when word wrap occurs--the new line has the same indentation level as the previous line--and hence the behavior of the reformatting commands (<CtrlB> and <AltR>).

EMwordWrap **<CtrlO><W>**
Toggle word wrap on or off. When word wrap is on, any attempt to insert or append text beyond the right margin causes a new line to be inserted following the current line and all words that are at least partially beyond the right margin to be moved to the new line.

EMreformatP **<CtrlB>**
Reformat the current paragraph.

EMreformatG **<AltR>**
Reformat the entire file. Use this command with caution.

EMrestore **<CtrlQ><L>**
Restore the original contents of the current line.

EMhelp **<F1>, <ClickBoth>**
Help. If a user-written help routine was established by setting MemoHelpPtr to a value other than nil, pressing <F1> calls that routine; otherwise this command does nothing.

EMquit **<Esc>, <CtrlBreak>, <ClickRight>**
Quit editing.

EMmouse **<ClickLeft>**
Move the cursor to the position indicated by the mouse.

In addition to the commands described above, TPMEMO allows up to 20 user-defined commands (EMuser0..EMuser19). See "Programming Hooks", below, for details.

Programming Hooks
TPMEMO implements many of the same kinds of hooks as TPEDIT, TPENTRY, and several of the other Turbo Professional units (see Appendix B). These hooks are:

MemoKeyPtr
This variable points to the routine used to obtain keyboard input. See

"Background Task Hooks" in Appendix B for details on how to use a pointer variable of this type.

MemoHelpPtr
This variable points to the routine to be called when a request for help is issued by the user. See "Context-Sensitive Help Hooks" in Appendix B for details on how to use a pointer variable of this type.

MemoKeyMax, MemoKeyID, and MemoKeySet
These constants together provide an installable keyboard hook, as defined in "Installable Keyboard Hooks" in Appendix B.

TPMEMO also provides a couple of hooks for which there is no real counterpart in any of the other Turbo Professional units. These hooks are:

MemoErrorPtr
This variable points to a routine that displays error messages when an error occurs. The routine must be of the form:

```
{$F+}
procedure ErrorRoutine(var EMCB : EMcontrolBlock; ErrorCode : Word);
begin
end;
{$F-}
```

The EMCB parameter is the current editor control block. ErrorCode indicates the type of error that occurred. The four possible values are:

```
tmBufferFull     = 1;        {edit buffer is full}
tmLineTooLong    = 2;        {line too long, CRLF inserted}
tmTooManyLines   = 3;        {max line limit exceeded}
tmOverLineLimit  = 4;        {max line limit already exceeded}
```

For an example of an error routine, see the MemoError procedure in TPMEMO.PAS. It is imperative that an error routine be declared FAR and that it not be nested inside another procedure.

MemoStatusPtr
This variable points to a routine that displays status information while editing is taking place. The routine must be of the form:

```
{$F+}
procedure StatusRoutine(var EMCB : EMcontrolBlock);
begin
end;
{$F-}
```

The one parameter passed to a status routine is the current editor control block. For an example of a status routine, see the MemoStatus procedure in TPMEMO.PAS. It is imperative that a status routine be declared FAR and that it not be nested inside another procedure.

UserData field in an EMcontrolBlock
The first field in a variable of type EMcontrolBlock is a pointer called 'UserData'. This space is set aside for you to use as you see fit. You could use it to store a pointer to a filename or a related data structure, or you might typecast the pointer to a longint and use it to store an ID number (this might be useful if you have multiple editing windows open or multiple memo fields in a database record):

```
var
  EMCB : EMcontrolBlock;
begin
  LongInt(EMCB.UserData) := 1;
  InitControlBlock(EMCB, ...);
end;
```

No routine in TPMEMO ever uses this field. It is provided strictly for your convenience in cases where you need it.

Compiling TPMEMO
Like many of the units in Turbo Professional, TPMEMO depends on conditional compilation directives that affect the size and capabilities of the unit. At present, TPMEMO depends on only a single conditional define in TPDEFINE.INC, 'UseMouse'. If UseMouse is defined, TPMEMO can use the mouse handling routines in TPMOUSE to provide built-in mouse support. (Note that you must call EnableMemoMouse in order to actually enable mouse support.) If you don't need mouse support, you should undefine UseMouse in TPDEFINE.INC before compiling TPMEMO.

Example Programs
Two programs, ENTRY.PAS and MEMO.PAS, demonstrate the use of TPMEMO. Together, these programs provide an ample demonstration of what can be done with TPMEMO.

To see TPMEMO in action, load ENTRY, move the cursor to the Notes field and press <Enter>. To get help concerning the use of the memo editor from within ENTRY, press <F1>.

MEMO.PAS is a simple text editor based on TPMEMO. To use it, enter

```
MEMO filename
```

where filename is the name of the file to be entered. In addition to the editing commands described above, MEMO implements three user-defined commands:

EMuser0 **<CtrlK><S>, <F2>**
Save the file and continue editing.

EMuser1 **<CtrlK><X>, <Ctrl F2>**
Save the file and exit.

EMuser2 **<CtrlK><Q>, <Alt F2>**
Abandon the file. If the file was modified, you are asked to confirm that you want to quit without saving the changes. As implemented, this command is equivalent to the EMquit command.

If you are at all familiar with the Turbo Pascal editor, you should find the behavior of MEMO quite intuitive. Keep in mind, however, that TPMEMO does not implement any of the block-related commands found in TURBO's editor.

Miscellaneous Notes

Common Mistakes

Probably the most common mistake people make when using TPMEMO is that of calling ReadMemoFile *after* they have called InitControlBlock. The problem with doing so is that ReadMemoFile knows nothing about the control block, and consequently it cannot reinitialize the control block based on the new contents of the buffer. There are two ways to insure that the control block is initialized properly: (1) Call ReadMemoFile *before* calling InitControlBlock, as is done in MEMO.PAS; or (2) call InitBufferState immediately following the call to ReadMemoFile.

The Modified Flag

The Modified flag in an editor control block indicates whether or not any changes have been made to the edit buffer. This flag is temporarily set any time that a change is made to the text of a line. If the changes are undone (either manually or using the restore line command, <CtrlQ><L>), the flag is reset to its previous state. Once the cursor is moved from a modified line, the flag remains set until it is cleared by a call to InitControlBlock or SaveMemoFile.

Reentrancy

The editor in TPMEMO is completely reentrant. That is, it does not alter any global variables other than the editor control block passed to it (which may or may not be a global variable). The main implication of this feature is that it is

possible for a user-written routine called by EditMemo to in turn call EditMemo to edit a second file or memo field. The screen manager in TPENTRY is also reentrant in the same sense.

Edit Window Size, Etc.
The minimum size of an edit window is 1 row x 1 column, although you would obviously never want a window that narrow. You might, in unusual cases, want a one-row window, however, because it is technically possible to use TPMEMO as a line editor (like TPEDIT). To do so, you would

- specify a one-row window (YLow = YHigh) in the call to InitControlBlock
- specify a LineLimit parameter of 1 in the call to InitControlBlock
- and, optionally, set MaxLineLength to the width of the edit window less 1 (be sure to reset MaxLineLength to 127 afterward)

This might be useful in cases where TPMEMO is needed in a program, and you want to avoid pulling in TPEDIT. Keep in mind, however, that <Enter> is not an exit command in TPMEMO. You would probably want to temporarily reassign <Enter> to an exit command such as EMuser0 if you are going to do this.

Tabs
TPMEMO inserts tabs by adding the appropriate number of spaces to a line. It is not equipped to handle "hard tabs" (ASCII 9).

Line Limit in Editor
Although the memo field editor allows you to specify a limit on the number of lines that the user may enter, that limit is not enforced in two cases: when the buffer contains too many lines before editing begins, and when text is being reformatted (there's no good way to determine in advance how many lines there will be after reformatting). In both cases, the editor simply generates the 'tmOverLineLimit' error.

TPMEMO: Declarations

Constants

```
AllowTruncation : Boolean = True;
```

This boolean constant determines whether or not ReadMemoFile will read in a partial file in cases where the file is larger than the edit buffer.

```
DeleteJoinsLines : Boolean = False;
```

If this constant is True, pressing <Del> when the cursor is beyond the end of the current line will pull up the following line, just as the delete word command (<CtrlT>) does.

```
DisallowedInReadOnlyMode : set of EMtype =
    [EMchar..EMenter, EMrestore, EMback..EMreformatG];
```

The commands in this set are disallowed in read-only mode. It is interfaced in case you need to add any user-defined commands to the set.

```
EMnone        = 00; {Not a command}
...
EMuser19      = 52;
```

Each of these constants corresponds to an editing, cursor movement, or exit command implemented by TPMEMO. See "Editing Commands", earlier in this section, for a discussion of these commands.

```
ErrorRow : Byte = 1;      {default to top line of screen for error messages}
ErrorAttr : Byte = $F;    {attribute for error message line}
```

These constants determine which row MemoError uses to display error messages, and which video attribute they are displayed in. They are not used unless you enable MemoError with the statement 'MemoErrorPtr := @MemoError;'.

```
HelpForMemo = HelpForXXXX1;
```

This constant represents the special code passed to a help routine to indicate which unit the request for help came from. See "Programming Hooks", earlier in this section.

```
IndentStartsParagraph : Boolean = False;
```

If this constant is True, TPMEMO's paragraph reformatting routine treats a blank space at the beginning of a line as signalling the start of a new paragraph and hence the end of the previous one. If it is False, the end of a paragraph is signalled by a blank line.

```
MaxLineLength : Byte = 127;
```

The maximum length of a line of text. This typed constant can be made smaller, but under no circumstances should it be made larger. (The word wrap routine needs to be able to store two complete lines of text in a single string variable.)

```
MemoErrorPtr : Pointer = nil;
```

If not nil, this variable points to a routine that is called each time an error occurs. See "Programming Hooks", earlier in this section.

```
MemoHelpPtr : Pointer = nil;
```

This variable points to a routine that displays help when <F1> is pressed. If it is nil, pressing <F1> does nothing. See "Programming Hooks", earlier in this section.

```
MemoKeyMax = 250;
MemoKeyID : string[16] = 'tpmemo key array';
MemoKeySet : array[0..MemoKeyMax] of Byte = (...);
```

MemoKeySet contains the default key assignments for all the commands. See "Programming Hooks", earlier in this section.

```
MemoKeyPtr : Pointer = nil;
```

This variable points to a routine that returns the next keystroke. By default it points to either TPMOUSE.ReadKeyOrButton (if mouse support is enabled) or TPCRT.ReadKeyWord (if it isn't). See "Programming Hooks", earlier in this section.

```
{$IFDEF UseMouse}
MemoMouseEnabled : Boolean = False;
```

This typed constant is True if mouse support was enabled by a call to EnableMemoMouse.

```
MemoStatusPtr : Pointer = nil;
```

If not nil, this variable points to a routine that is called at each keystroke. Normally this routine displays a status line. See "Programming Hooks", earlier in this section.

```
StatusRow : Byte = 2;        {default to second line of screen for status line}
StatusAttr : Byte = $F;      {attribute for status line}
```

These constants determine which row MemoStatus uses to display a status line, and which video attribute the status line is displayed in. They are not used unless you enable MemoStatus with the statement 'MemoStatusPtr := @MemoStatus;'.

```
tmBufferFull     = 1;        {edit buffer is full}
tmLineTooLong    = 2;        {line too long, CRLF inserted}
tmTooManyLines   = 3;        {max line limit has been reached}
tmOverLineLimit  = 4;        {max line limit has been exceeded}
```

These constants represent the possible error message codes passed to a user-written error handler. See "Programming Hooks", earlier in this section.

TPMEMO: Declarations

Types

```
EMbuffer = array[1..65521] of Char;
EMcontrolBlock =                    {an editor control block}
  record
    UserData : Pointer;             {reserved for user (ID number perhaps)}
    XL, YL, XH, YH : Byte;          {coordinates for edit window}
    BufPtr : ^EMbuffer;             {pointer to text buffer}
    BufSize : Word;                 {size of buffer}
    MaxLines : Integer;             {maximum number of lines}
    TotalBytes : Word;              {bytes in buffer}
    TotalLines : Integer;           {lines in buffer}
    LineAtTop : Integer;            {line at top of edit window}
    BufPosTop : Word;               {index into buffer for start of line at top}
    CurLine : Integer;              {line number of current line}
    BufPos : Word;                  {index into buffer for start of current line}
    CurCol : Byte;                  {position of cursor within current line}
    ColDelta : Byte;                {for horizontal scrolling}
    KnownLine : Integer;            {used to speed up scrolling/searching}
    KnownOfs : Word;                {"    "  "     "  "}
    TAttr : Byte;                   {attribute for normal text}
    CAttr : Byte;                   {attribute for control characters}
    InsertMode : Boolean;           {True if in insert mode}
    IndentMode : Boolean;           {True if in auto-indent mode}
    WordWrap : Boolean;             {True if word wrap is on}
    Modified : Boolean;             {True if edits have been made}
    TabDelta : Byte;                {distance between tab stops}
    Margin : Byte;                  {right margin}
    HelpTopic : Word;               {help topic}
  end;
```

Variables of this type are used to store information about a text editing window and its associated buffer.

```
EMtype = EMnone..EMuser19;
```

A variable of this type is generally used to store the result of a call to EditMemo, which always returns a value in the range EMnone..EMuser19.

```
MemoStatusType = (
  mstOK, mstNotFound, mstInvalidName, mstReadError, mstTooLarge,
  mstTruncated, mstCreationError, mstWriteError, mstCloseError);
```

ReadMemoFile and SaveMemoFile both return a function result of this type. mstOK indicates that no error occurred. ReadMemoFile returns mstNotFound if the specified file does not exist. It is not an error code; it simply indicates that the user wants to edit a file that does not yet exist. mstInvalidName is returned by ReadMemoFile if the drive and/or directory in the specified filename is invalid. mstReadError is returned by ReadMemoFile if an error occurred while reading in the text file. mstTooLarge is returned by ReadMemoFile if the file to be read is larger than the edit buffer and AllowTruncation is False; if AllowTruncation is True, as it is by default, mstTruncated is returned instead. mstCreationError is returned by SaveMemoFile if it is unable to create the specified file (because the disk is full or is write-protected, for example). mstWriteError is returned by SaveMemoFile if an error occurs while writing the edit buffer contents to disk. mstCloseError indicates that an error occurred while attempting to close a file.

Routines

The following routines are intended primarily for internal use, but they are interfaced in case you need them to implement user-defined commands or for other purposes. These routines are not fully documented, so if you want to use one of them, study the source code for TPMEMO.

```
function FindLineIndex(var EMCB : EMcontrolBlock; LineNum : Integer) : Word;
  {-Return the index into the edit buffer for the specified line number.
    LineNum must be <= EMCB.TotalLines.}

function FindLineLength(var EMCB : EMcontrolBlock; LineNum : Integer) : Word;
  {-Find the length of the specified line}

procedure InitBufferState(var EMCB : EMcontrolBlock; BufferSize : Word;
                          var Buffer);
  {-Initialize the edit buffer status fields in a control block}

procedure GetLine(var EMCB : EMcontrolBlock; var S : string; LineNum : Integer);
  {-Get the LineNum'th line from the buffer for the specified control block and
    store it in S. If line is longer than 255 characters, only the first 255
    characters will be loaded into S.}

procedure DrawLine(var EMCB : EMcontrolBlock; St : String; LineNum : Integer);
  {-Draw the string St, which represents the specified line number}

procedure FastWriteCtrl(St : String; Row, Col, Attr, Ctrl : Byte);
  {-Write St at Row,Col in Attr (video attribute) without snow. Control
    characters displayed in Ctrl appear as upper-case letters}
```

AddMemoCommand / DisableMemoMouse

Declaration

```
function AddMemoCommand(Cmd : EMtype; NumKeys : Byte;
                        Key1, Key2 : Word) : Boolean;
```

Purpose

Add a new command key assignment or change an existing one.

Comments

This routine can be used to assign a user-defined command to a key combination, to disable an existing command, or to change a key assignment. See MEMO.PAS for examples of how to use this routine. For a general discussion of how to use a function of this type, see Appendix B.

Declaration

```
{$IFDEF UseMouse}
procedure DisableMemoMouse;
```

Purpose

Disable mouse support in TPMEMO.

Comments

This routine disables mouse support in TPMEMO. Note that this routine is not included in TPMEMO unless UseMouse is defined.

See Also

EnableMemoMouse

Declaration

```
function EditMemo(var EMCB : EMcontrolBlock; ReadOnly : Boolean;
                  var CmdList) : EMtype;
```

Purpose

Edit a buffer filled with text.

Comments

The main routine in TPMEMO, EditMemo is the function that processes all editing and cursor movement commands. On exit it returns a code indicating which command was used to exit the editor (usually EMquit, but possibly a user-defined command).

EMCB is a variable containing all pertinent information about an edit window and the text buffer associated with it. It must have been initialized previously by a call to InitControlBlock.

ReadOnly is used to select read-only mode, in which only non-destructive commands are processed. Although this option allows TPMEMO to be used as a file browser, the primary reason for providing it is to complement the read-only mode in TPENTRY.

Though untyped, CmdList should generally be a variable of the form

```
MyCmdList : array[1..n] of EMtype;
```

This array contains a list of commands to be processed by EditMemo immediately upon entry to the editor. For example, if you wanted to simulate the pressing of <Ctrl PgDn>, you could pass the following variable as the CmdList parameter:

```
MyCmdList : array[1..2] of EMtype = (EMendOfFile, EMnone);
```

The EMendOfFile command would tell EditMemo to move the cursor to the end of the file. The EMnone command signals the end of the list of commands and tells EditMemo to resume processing commands entered at the keyboard.

In some cases you may find it necessary to pass text as well as commands. To do so, simply pass a data structure containing the characters to be entered, surrounded by the value EMchar. For example:

```
MyCmdList : array[1..n] of Char =
  (Char(EMchar), 'h', 'e', 'l', 'l', 'o', Char(EMchar), Char(EMnone));
```

If you pass this typed constant as the CmdList parameter, the word 'hello' is inserted into the text buffer. The first EMchar signals the start of a series of characters to be entered into the text buffer. The second EMchar signals the end of the series. And the final EMnone signals the last command in the list. (Note that, if you are using Turbo Pascal 4.0, you need to change 'Char(EMchar)' to '#1' and 'Char(EMnone)' to #0 in the example above.) When using this facility, please do not try to pass control characters as part of a sequence of text.

Two final notes about this routine. First, EditMemo does not 'home the cursor' on entry. If an exit command is given and you call EditMemo again without calling InitControlBlock first, the cursor remains where it was before the exit command

was given. Although this behavior may be undesirable in some cases, it is necessary to allow certain kinds of user-defined commands to be implemented. Second, EditMemo does not preserve and restore the contents of the screen beneath the edit window. It is up to you to do so if necessary.

Example

```
var
  MyEMCB : EMCB;
  ExitCommand : EMtype;
const
  MyCmdList : EMtype = EMnone;
begin
  InitControlBlock(MyEMCB, ...);
  ...
  ExitCommand := EditMemo(MyEMCB, False, MyCmdList);
end;
```

Edit the text buffer associated with MyEMCB. No special edit commands are to be executed on entry to the editor. See MEMO.PAS and ENTRY.PAS for better examples of how to use EditMemo.

See Also

InitControlBlock

EnableMemoMouse / InitControlBlock

Declaration

```
{$IFDEF UseMouse}
procedure EnableMemoMouse;
```

Purpose

Enable mouse support in TPMEMO.

Comments

This routine enables mouse support in TPMEMO. Note that this routine is not included in TPMEMO unless UseMouse is defined.

Declaration

```
procedure InitControlBlock(var EMCB : EMcontrolBlock;
                           XLow, YLow, XHigh, YHigh : Byte;
                           TextAttr, CtrlAttr : Byte;
                           InsertOn, IndentOn, WordWrapOn : Boolean;
                           TabSize : Byte; HelpIndex : Word;
                           RightMargin : Byte; LineLimit : Integer;
                           BufferSize : Word; var Buffer);
```

Purpose

Initialize a memo editor control block.

Comments

This routine initializes an "editor control block", a variable that contains all pertinent information about an edit window.

EMCB is the control block to be initialized. XLow, YLow, XHigh, and YHigh give the coordinates for the edit window. TextAttr is the video attribute used to display normal text. CtrlAttr is the video attribute used to display control characters; if it is the same as TextAttr, the command that allows control characters to be entered (EMctrlChar, normally assigned to <Ctrl P>) is disabled. InsertOn indicates whether insert mode should be On by default. IndentOn indicates whether auto-indent mode should be On by default. WordWrapOn indicates whether or not word wrap should be enabled by default. TabSize specifies the distance between tab stops. HelpIndex is the value that should be passed to a help routine when <F1> is pressed. RightMargin indicates the column at which word wrap should occur. LineLimit indicates the maximum number of lines that the user may enter. Buffer is the edit buffer associated with EMCB, and BufferSize indicates its size.

InitControlBlock does three things. First, it initializes EMCB with the values specified by XLow and the other parameters. Second, it scans the contents of the edit buffer to determine how many bytes are in it and how many lines of text it contains. (The end of a line is signalled by a carriage return-line feed combination; the end of the edit buffer by a ^Z.) Third, it resets a few housekeeping variables that are used internally to keep track of the cursor, the current line, etc.

Example

See MEMO.PAS and ENTRY.PAS for examples.

MemoError / MemoStatus

Declaration

```
procedure MemoError(var EMCB : EMcontrolBlock; ErrorCode : Word);
```

Purpose

Display an error message and wait for a key press.

Comments

This routine provides a default error handler for TPMEMO. You can use it as is, or use it as a model for writing your own error handler. MemoError displays error messages on the row indicated by ErrorRow in the video attribute indicated by ErrorAttr.

Example

```
ErrorRow := 1;
ErrorAttr := $F;
MemoErrorPtr := @MemoError;
```

Enables MemoError and requests that error messages be displayed on row 1 in white on black.

Declaration

```
procedure MemoStatus(var EMCB : EMcontrolBlock);
```

Purpose

Display a status line.

Comments

This routine provides a default status display routine for TPMEMO. You can use it as is, or use it as a model for writing your own status display routine. MemoStatus displays its status line on the row indicated by StatusRow in the video attribute indicated by StatusAttr.

Example

```
StatusRow := 2;
StatusAttr := $F;
MemoStatusPtr := @MemoStatus;
```

Enables MemoStatus and requests that the status line be displayed on row 2 in white on black.

Declaration

```
function ReadMemoFile(var Buffer; BufferSize : Word; FName : string;
                      var FSize : LongInt) : MemoStatusType;
```

Purpose

Read a file into Buffer, returning a status code.

Comments

This routine reads a text file, FName, into a Buffer of BufferSize bytes. Upon return, FSize contains the size of FName, if it was found, and the function result indicates whether or not the operation was successful.

Be sure to read the section entitled "Common Mistakes" earlier in this section for information about when to call ReadMemoFile.

Example

```
MemoStatus :=
  ReadMemoFile(MyEditBuffer^, MyBufferSize, 'C:\AUTOEXEC.BAT', FSize);
```

Read C:\AUTOEXEC.BAT into the buffer pointed to by MyEditBuffer. MyBufferSize has the size of the edit buffer. On return, FSize has the size of the file being edited, if found.

Declaration

```
function SaveMemoFile(var EMCB : EMcontrolBlock; FName : string;
                      MakeBackup : Boolean) : MemoStatusType;
```

Purpose

Save the current contents of the edit buffer in a text file.

Comments

This routine saves the contents of the edit buffer associated with EMCB in the text file designated by FName. If FName already exists and MakeBackup is True, the existing FName is renamed to FName.BAK before the new file is created. Upon return, the function result indicates whether or not the operation was successful.

Example

```
MemoStatus := SaveMemoFile(MyEMCB, MyFileName, True);
```

Save the contents of the edit buffer associated with MyEMCB in MyFileName, and make a backup file if appropriate.

Identifiers

A

AbsBCD 668
ActionCodeType 489
ActivateTiledWindow 88
ActivateVScreen 89
ActivateWindow 91
ActivePage 265
AddArrayField 324
AddBackSlash 424
AddBCD 668
AddBcdField 325
AddBooleanField 326
AddByteField 327
AddCharField 327
AddChoiceField 328
AddCompField 329
AddDateField 330
AddDateStField 331
AddDblField 332
AddExtField 332
AddFieldPrim 317
AddIntField 333
AddLongField 333
AddMemoCommand 414
AddNestedField 334
AddRealField 335
AddSglField 335
AddShortField 336
AddStringField 337
AddTimeField 338
AddUserField 339
AddWindowToLibrary 32
AddWordField 340
AddYesNoField 340
AllocateEmsPages 591
AllocateMacro 613
AllocateScreen 266
AllocateStack 531
AllowPendingSubmenus 117
AllowTruncation 410
AltPickAttr 238
ArcTanBCD 669
Asc2Str 466
AscCenter 466
AscCenterCh 467
AscCharStr 467
AscFromHeap 468
Asciiz 465
AsciizPtr 465
AscLeftPad 469
AscLeftPadCh 469
AscLoCase 470
AscPad 471
AscPadCh 471
AscToHeap 472
AscTrim 473
AscTrimLead 473
AscTrimTrail 474
AscUpCase 474
AssignCrt 25, 266
AssignMacroName 614
AuxKeyLength 612

B

BadDate 381
BadTime 381
BCD 667
BCDsize 667
BCDtoReal 669
BinaryB 424
BinaryL 425
BinaryW 425
BiosScroll 28
Black 24, 264
Blink 24
BlockCursor 32
BlockMouseCursor 212
Blue 24, 264
BMMakeTable 426

BMSearch 427
BMSearchUC 428
Brown 24, 264
BTable 423
BufLen 30
ButtonStatus 210
BW40 24
BW80 24

C

C040 24
C40 24
C80 24
CallOldIsr 693
CancelAllPrintFiles 490
CancelPrintFile 490
CanRecover 689
CaseChange 319
Center 429
CenterCh 429
CenterWrite 267
ChainInt 532
ChangeAttribute 33
ChangeAttributeWindow 33
ChangeDirectory 256
ChangeFieldAttr 341
ChangeHotKey 554
ChangePromptAttr 341
ChangeProtection 342
ChangeProtectionFast 343
ChangeRequired 343
ChangeStringAttr 344
ChangeValidation 344
CharSet 159, 319, 423
CharStr 430
CharToMacro 614
CheckBreak 25
CheckEOF 25
CheckKbd 34
CheckMenuStatus 121
CheckSnow 25
CleanPathName 430
ClearA 633, 645, 654
ClearEntryWindow 345
ClearFirstChar 157
ClearFlag 693
ClearMacros 615
ClearVScreen 91
ClearWindow 116
CloseHelp 182
CloseLibrary 34
ClrEol 25, 262
ClrScr 25, 262
CO80 24
ColonChar 381
ColorColors 267
CommaChar 320
CommaForPeriod 423, 667
CommandSet 319
CompareLetterSets 431
CompareType 423
CompAsc 475
CompString 433
CompStruct 434
CompUCAsc 475
CompUCString 435
ConcatAsc 476
ConcatStr 476
CopyAsc 477
CopyScreen 268
CopyVirtualScreenToWindow 317
CopyWindow 269
CosBCD 670
CreateLibrary 35
CrtError 28
CtrlBreakFlag 30
CurrencyLtStr 320
CurrencyRtStr 320
CurrentCols 271
CurrentDisplay 30
CurrentFieldModified 345
CurrentMode 30
CurrentPage 30

CurrentRows 271
CurrentScreen 265
CurrentTime 385
CurrentTimeString 385
CurrentWindow 87
CursorEndLine 36
CursorStartLine 36
CursorToEnd 157
CursorTypeSL 37
Cyan 24, 264

D

DarkGray 24, 264
Date 381
Date1900 382
Date1980 382
Date2000 382
DateDiff 386
DatePicture 254
DateString 381
DateStringToDate 386
DateStringToDMY 387
DateTimeDiff 388
DateTimeRec 381
DateToDateString 389
DateToDMY 389
DateToSortString 390
DayOfWeek 390
DayString 382
DayType 381
DeactivateVScreen 92
DeallocateEmsHandle 592
DeallocateMacro 615
DeallocateScreen 272
DeallocateStack 533
DecimalChar 320
DefaultDrive 491
DefaultExtension 435
DefaultMonth 384
DefaultYear 384
DefinedKeys 612
DefineMacro 616
DefinePop 555
DefinePopProc 556
DefineScreen 272
Delay 25
DeleteAsc 477
DeleteJoinsLines 410
DeletePop 557
DeletePopProc 557
DeleteWindowFromLibrary 38
DelLine 25
deName 27
DeTab 436
DetectMultitasking 29
DirDisplayStr 254
DirectoryEntry 27
DirectoryPtr 27
DirectoryType 27
DirectVideo 25
DirsUpper 254
DisableEditMouse 160
DisableEntryMouse 346
DisableEventHandling 212
DisableHelpIndex 182
DisableHelpMouse 183
DisableMemoMouse 414
DisableMenuItem 122
DisableMenuMouse 124
DisablePickMouse 242
DisableTSR 558
DisallowedInReadOnlyMode 410
DiskClass 489
DisplayOverride 29
DisplayType 27
DisplayWindow 93
DisposeA 633, 645, 654
DisposeAsc 478
DisposeEditScreen 346
DisposeMenu 124
DisposePackedWindow 40
DisposeString 437
DisposeSubMenu 116

DisposeVScreen 94
DisposeWindow 94
DispPackedWindow 39
DispPackedWindowAt 39
DivBCD 670
DMYtoDate 391
DMYtoDateString 391
DosBusyFlag 559
DosCriticalFlag 559
DOSVersion 491
DrawEditScreen 347
DrawField 347
DrawFrame 116
DrawItem 116
DrawLine 413
DrawMenuFromStack 125
DrawShadowFrame 85
DrawSubMenu 116
Dummy5 530

E

EditButtonCodes 159
EditHelpIndex 159
EditHelpPtr 159
EditKeyPtr 159
EditKeys 627
EditMapCtrls 157
EditMemo 415
EditScreen 348
EditSize 157
EditString 349
ElementsSorted 582
EMcontrolBlock 412
EmsActiveHandles 593
EmsAvailable 644
EmsErrorCode 590
EmsInstalled 593
EmsPageFramePtr 594
EmsPagesAvail 595
EmsPagesOwned 596
EmsStatusOK 597
EmsTotalPages 598
EmsVersion 598
EMtype 412
EmulateInt 534
EnableEditMouse 160
EnableEnhanced 522
EnableEntryMouse 351
EnableEventHandling 213
EnableHelpIndex 183
EnableHelpMouse 184
EnableMemoMouse 417
EnableMenuItem 125
EnableMenuMouse 126
EnablePickMouse 243
EndOfMacro 611
EndPrintStatus 492
EnhancedDisplay 30
EnTab 438
EntryHelpPtr 320
EntryKeyPtr 323
EntryMapCtrls 320
EqualBCD 671
EraseCurrentSubMenu 128
EraseMenu 129
EraseMenuOntoStack 130
EraseSubMenu 116
EraseTopWindow 95
Error8087 570
ErrorA 634, 646, 655
ErrorAttr 410
ErrorRecovery 690
ErrorRow 410
ESrecord 319
EStype 319
EvaluateEScommand 352
Exceptions8087 571
ExchangeBytes 694
ExchangeStructs 694
ExchangeWords 695
ExecDos 493
ExistFile 495
ExistOnPath 496

ExpBCD 672
Explode 86
ExplodeDelay 86
ExplodeFrame 85
ExtMemAvail 603
ExtMemTotal 603
ExtractWord 439

F

FarCall 695
FastCenter 40
FastFill 41
FastFillWindow 41
FastFlush 42
FastRead 43
FastReadLine 273
FastReadWindow 43
FastText 44
FastTextWindow 44
FastVert 45
FastVertWindow 45
FastWrite 46
FastWriteClip 96
FastWriteCtrl 413
FastWriteMem 273
FastWriteStack 97
FastWriteWindow 47
FatCursor 47
FatMouseCursor 213
FieldRec 319
FileHandlesLeft 497
FileHandlesOpen 497
FilesUpper 254
FillAttribChar 274
FillPickWindow 245
FillStruct 696
FillVideoChar 274
FillWindow 275
FillWord 696
FiltersEnhanced 522
FindFieldID 317
FindHelp 186
FindLineIndex 413
FindLineLength 413
FindMacroIndex 616
FindMenuItem 131
FlagIsSet 697
FlexAttrs 27
FlexWrite 48
FlexWriteWindow 49
Float 423
FlushA 657
Font8x8 24
Font8x8Selected 49
ForceCaseUser 321
ForceExtension 439
ForceUpper 157
Form 440, 672
FracBCD 673
FrameArray 27
FrameChars 29
FrameCharType 27
FrameDelta 117
FrameWindow 50, 263
FreeExtMem 604
FullMouseWindow 214
FullPathName 442

G

GetCrtMode 50
GetCurrentMenuKey 131
GetCursorState 51
GetDiskClass 498
GetDiskInfo 499
GetDta 500
GetElement 583
GetEnvironmentString 501
GetExtMem 604
GetFileMode 502
GetFileName 257
GetLine 413
GetMenuItemName 132

GetMickeyCount 214
GetMousePage 215
GetPrintStatus 503
GoToXY 25, 262
GotoXYAbs 51, 263
GreaterBCD 673
GreaterEqualBCD 674
Green 24, 264

H

HaltOnError 632, 644, 653
HandleIsConsole 504
HardMouseCursor 216
HasEnhancedKbd 522
HelpCmdNum 181
HelpColorArray 179
HelpColorType 179
HelpForMemo 410
HelpKeyPtr 181
HelpMore 180
HelpMouseScroll 181
HelpPtr 179
HelpTitle 180
HercCardType 28
HercGraphicsMode 52
HercModeTestWorks 52
HercPresent 53
HexB 443
HexL 443
HexPtr 444
HexW 444
HiddenCursor 54
HiddenMouseCursor 217
HideCursor 117, 181, 238
HideCursorInReadChar 157
HideMouse 217
HighVideo 25, 262
HiWord 697
HKtype 179
HMStoTime 392
HoursInDay 383
HouseCursorAtEnd 157
HugePtr 602

I

IfcPtr 552
IfcRecord 552
ImplodeFrame 85
IncDate 392
IncDateTime 393
IncTime 393
IndentStartsParagraph 410
InitBufferState 413
InitControlBlock 417
InitESrecord 353
InitializeMouse 218
InitKbdVectors 523
InitMacros 617
InitPictureFlags 317
InitPops 560
InitType 632, 644, 653
InitVector 535
InitVideo 275
InsertAsc 479
InsertByDefault 157
InsertMode 321
InsertStr 479
InsLine 25
Installed8087 572
InstallModule 560
Int24Off 576
Int24On 576
Int24Result 561, 577
IntBCD 674
InternationalCurrency 354
InternationalDate 394
InternationalTime 395
InterruptsOff 536
InterruptsOn 536
InTextMode 30
IntnnHandle 552
IntRegisters 530

IntReturn 537
IoResultPtr 489
ISR_Array 530
ISR_Record 530

J

JumpRecord 692
JumpToOldIsr 698
JustExtension 445
JustFilename 445
JustPathname 446

K

KbdFlags 54
KeepDiskCurrent 653
KeyPressed 25
KeyToString 617
KillTiledWindow 98
KillWindow 98

L

LastEntryCommand 323
LastMode 25
LeftPad 447
LeftPadCh 447
LenAsc 480
LessBCD 675
LessEqualBCD 675
LetterValues 423
LibName 27
LightBlue 24, 264
LightCyan 24, 264
LightGray 24, 264
LightGreen 24, 264
LightMagenta 24, 264
LightRed 24, 264
LnBCD 676
LoadA 636, 648, 658
LoadVA 637
LoCase 448
LoCaseMac 448
Long2Str 449
LongintToBCD 676
LongJump 699
LotusFrame 117
LoWord 700
LowVideo 25, 262

M

MacEdKeypressedPtr 626
MacroAddress 618
MacroCount 618
MacroFileID 611
MacroInProgress 612
MacroNames 612
MacroPointers 612
MacroRec 611
MacroRecording 612
MacroRecordingOff 619
MacroRecordingOn 619
MacroRecPtr 611
MacrosAreOn 612
MacrosOff 620
MacrosOn 620
Magenta 24, 264
MakeA 638, 649, 659
MakeAPrimitive 639, 661
MakeLetterSet 449
MakeVScreen 99
MakeWindow 100
MapColor 55
MapColors 29
MapEmsPage 599
MapPackedWindowColors 56
MarkExtMem 605
MaxAsciiz 465
MaxDate 383
MaxISRs 530
MaxKeysInMacro 611
MaxLineLength 410

MaxMacros 611
MaxPopups 552
MaxScreens 264
MaxSearchLen 238
MaxTime 383
MaxWord 700
MaxYear 383
MemoError 418
MemoErrorPtr 410
MemoHelpPtr 411
MemoKeyPtr 411
MemoKeySet 411
MemoMouseEnabled 411
MemoStatus 418
MemoStatusPtr 411
MemoStatusType 412
Menu 119
MenuCharSet 119
MenuChoice 133
MenuCmdNum 120
MenuColorArray 119
MenuColorType 119
MenuGetKeyPtr 120
MenuHeight 135
MenuHelpPtr 120
MenuItem 136
MenuItemProc 117
MenuItemPtr 137
MenuKey 119
MenuMode 138
MenuStatus 139
MenuWidth 139
MergePicture 317
MinDate 383
MinSpaceForDos 489
MinTime 383
MinutesInHour 383
MinWord 701
MinYear 383
MKType 120
ModuleInstalled 561
ModulePtrByName 562
MoneySign 423, 667
Mono 24
MonoColors 276
MonthString 383
MouseButtonPressed 219
MouseButtonReleased 219
MouseCursorOn 211
MouseEventType 210
MouseGotoXY 220
MouseInstalled 211
MouseInWindow 220
MouseKeyWord 221
MouseKeyWordX 211
MousePressed 222
MouseStateBufferSize 222
MouseStatePtr 210
MouseWhereX 223
MouseWhereXY 223
MouseWhereY 224
MouseWindow 224
MouseXLo 210
MoveExtMem 606
MoveScreen 57
MoveVScreenToWindow 102
MoveWindow 104
MoveWindowHorizontal 277
MoveWindowToVScreen 105
MoveWindowVertical 278
MultBCD 677
MultiTaskingOn 30

N

NearCall 701
NewMenu 140
NoFrame 117
NoHelp 118
NonMacroKey 626
NormalAttr 30
NormalCursor 58
Normalized 702
NormalMouseCursor 225

NormVideo 25, 262
NoSound 25
NotEqualBCD 677
NotFound 465
NumberOfDrives 504

O

OctalB 450
OctalL 450
OctalW 451
OneMS 31
OpenHelpFile 187
OpenHelpMem 189
OpenLibrary 58
OpenStdDev 505
Orientation 120
OutputToScreen 265

P

PackedScreen 27
PackedWindow 28
PackedWindowPtr 28
PackLibrary 59
PackMenuLib 142
PackWindow 60
Pad 451
PadCh 452
ParagraphsToKeep 562
ParsePath 505
PasswordChar 157
PasswordMode 158
PiBcd 667
PickAlterPageRow 238
PickBar 247
PickChar 241
PickCmdNum 241
PickColorArray 237
PickColorType 237
PickHelp 191
PickHelpPtr 241
PickKeyPtr 241
PickMatrix 238
PickMaxRows 239
PickMinRows 239
PickMore 239
PickMousePage 239
PickMouseScroll 239
PickMouseWindow 239
PickOrientation 247
PickSrch 240
PickSrchStat 240
PickStick 240
PickUserPtr 241
PickWindow 248
PKType 237
PlaybackDelay 611
PopSubLevel 143
PopSubMenu 116
PopTickerPtr 552
PopupInUse 552
PopupsEnabledPtr 553
PopupsOff 563
PopupsOn 563
PopupToCallPtr 553
PosAsc 480
PosStr 481
PrintInstalled 506
PtrToHuge 607
PushSubMenu 116
PutElement 584

R

RangeCheck 632, 644, 653
ReadAttrAtCursor 61
ReadAttribute 62
ReadAttributeWindow 62
ReadCharacter 161
ReadCharAtCursor 63
ReadDiskSectors 507
ReadInteger 162
ReadKey 25

ReadKeyOrButton 225
ReadKeyWord 64
ReadLnAsc 482
ReadLongInt 163
ReadMacroFile 621
ReadMemoFile 419
ReadMenuLib 144
ReadPackedWindow 65
ReadReal 164
ReadStr 485
ReadString 165
ReadVScreen 106
ReadWindowFromLibrary 66
ReadWord 166
Real2Str 452
RealToBCD 678
Red 24, 264
RedrawMenu 145
RegKeyLength 612
Reinit8087 572
ReinitCrt 67
ReinitEmulator 573
ReinitScreen 279
ReleaseExtMem 607
ReleaseVideo 280
RemoveMacros 621
RemovePops 564
ResetEntryScreenFlags 355
ResetMenu 146
ResizeWindow 107
Restore8087 573
RestoreAllVectors 538
RestoreCursorState 67
RestoreEmsContext 600
RestoreKbdVectors 523
RestoreMouseState 226
RestoreShadowFrame 85
RestoreVector 538
RestoreWindow 68
RestoreWindowCoordinates 68, 281
RetA 640, 650, 662
ReturnCompletePath 255
ReturnStr 485
RingBell 355
RoundBCD 679
RoundToNearestHour 395
RoundToNearestMinute 396
RSChWord 159
RSCommand 159
RSpadChar 158
RStype 159

S

Save8087 574
SaveBuffer8087 569
SaveEmsContext 601
SaveInt1B 25, 31
SaveMemoFile 419
SaveMouseState 226
SaveShadowFrame 85
SaveWindow 69
ScrapMacro 612
ScrapMacroKey 612
ScrapMacroName 612
ScreenHeight 31
ScreenHelp 192
ScreenRec 264
Screens 265
ScreenWidth 31
ScrollByPage 323
ScrollWindow 108
ScrollWindowDown 70, 263
ScrollWindowUp 71, 263
Search 453
SearchUC 454
SecondsInDay 383
SecondsInHour 383
SecondsInMinute 383
SelectDrive 509
SelectFont8x8 72
SelectMenuItem 146
SelectOnFirstMouseClick 118
SelectScreen 281

SendEOI 539
SeparateDirs 254
SetA 641, 651, 663
SetActivePage 282
SetAllFieldLinks 356
SetAutoAdvance 358
SetAutoNumLock 359
SetBackgroundChar 282
SetBeepOnError 359
SetBell 360
SetBlink 73
SetBlock 509
SetClearFirstChar 360
SetCrtBorder 73
SetCtrlAttr 361
SetCursorSize 74
SetCursorToEnd 361
SetDelimiters 362
SetDta 510
SetEntryWindow 362
SetErrorPtr 363
SetExitOnSecondClick 364
SetFieldAttr 364
SetFieldLinks 365
SetFlag 702
SetForceMode 366
SetFrameChars 74
SetHelpPos 194
SetHercMode 75
SetHorizontalPick 250
SetInactiveFrame 108
SetInsertPushes 366
SetJump 703
SetMenuDelay 147
SetMenuItemName 148
SetMenuMode 149
SetMenuSelectKeys 150
SetMickeyToPixelRatio 227
SetMouseEventHandler 228
SetMousePage 229
SetNumeric 367
SetPadChar 367
SetPasswordChar 368
SetPasswordMode 368
SetPopTicker 544, 564
SetPostEditPtr 369
SetPreEditPtr 370
SetPromptAttr 370
SetProtectAttrs 371
SetProtection 371
SetRawMode 511
SetRequired 372
SetRightJustify 372
SetSnakingPick 250
SetStringAttr 373
SetTopTiledWindow 109
SetTopWindow 110
SetTrimBlanks 373
SetUpdatePtr 374
SetVerify 512
SetVerticalPick 251
SetVisiblePage 75
SetWaitCount 622
SetWrapMode 375
Shadow 86
ShadowAttr 87
ShadowMode 87
ShadowType 86
ShowExtension 254
ShowHelp 195
ShowMouse 229
ShowPrevHelp 198
ShowReadChar 158
ShowSizeDateTime 254
SideKickLoaded 552
SinBCD 680
SizeDisplay 254
SlashChar 383
SoftMouseCursor 230
Sort 585
SortStatus 582
SortStringToDate 396
SortStringToTime 397
Sound 25

Soundex 455
SoundFlagW 87
SpeedWrite 283
SqrBCD 681
SqrtBCD 681
SrchStart 240
SrchString 237
SrchType 237
StackSafetyMargin 489
StartMacro 622
StartMacroByKey 623
StatusAttr 411
StatusRow 411
StdErrHandle 489
StdInHandle 489
StdOutHandle 489
StdPrnHandle 489
StLoCase 455
StoreA 642, 652, 664
StoreVA 643
StoreWindowCoordinates 76, 284
Str2Asc 483
Str2Int 456
Str2Long 456
Str2Real 457
Str2Word 457
StrBCD 682
StrExpBCD 683
StringFromHeap 458
StringPointer 611
StringToHeap 459
StringToMacro 624
StringToScrapMacro 624
StripPicture 317
StuffKey 77
StuffString 77
StUpCase 460
SubBCD 683
SubMenu 151
SubmitPrintFile 513
SubstChars 323
SwapNibble 704
SwapStackAndCall 540
SwapStackAndCallNear 541
SwapWord 704
SwitchInColorCard 78

T

TempMacro 626
TerminateAndStayResident 565
TextAttr 25
TextBackground 25, 262
TextChar 30
TextColor 25, 262
TextFileSize 514
TextFlush 514
TextMode 25, 262
TextPos 515
TextSeek 515
ThisIfc 553
Threshold2000 384
Time 381
TimeDiff 397
TimeMs 516
TimePicture 254
TimeStringToTime 398
TimeToAmPmString 398
TimeToHMS 399
TimeToSortString 399
TimeToTimeString 400
Today 400
TodayString 401
TpArray 632, 644, 653
TpEntryScrolls 309
TPStr 484
Trim 460
TrimBlanks 158
TrimLead 461
TrimTrail 461
TruncBCD 684
Turbo3StyleColors 264

U

UninstallModule 565
UpCase 462
UpCaseMac 462
UpCaseTime 384
UseCrt 23
UseEmulator 574
UseFileFrame 255
UseHelpFrame 180

V

ValBCD 685
ValidateNoBlanks 376
ValidateNotPartial 376
ValidateSubfields 377
ValidDate 401
VerifyOn 516
VideoSegment 31
VirtualSegment 31
VirtualWidth 31
VisiblePage 265
VScreen 86
VScreenCols 111
VScreenRows 111

W

WaitForButtonRelease 211
WhereX 26, 262
WhereXAbs 79, 263
WhereXY 79, 263
WhereXYdirect 80
WhereY 26, 262
WhereYAbs 81, 263
WhichHerc 31
White 24, 264
WindMax 25
WindMin 25
Window 26, 262
WindowCoordinates 28, 264
WindowIsActive 112
WindowPtr 86
WindowRelative 158
WordCount 463
WordWrap 464
WrapModeType 320
WriteAsc 483
WriteAttribute 81
WriteAttributeWindow 82
WriteDiskSectors 517
WriteError 632, 644, 653
WriteMacroFile 625
WriteMenuLib 152
WritePackedWindow 82
WriteVScreen 113

Y

Yellow 24, 264
YesOrNo 167

Z

ZeroBcd 667

IDENTIFIERS

Index

0

$$WORK$$.LBR 59
<CtrlBreak> 30, 545, 551
<NumLock> 710
{$F+} 6, 690, 695, 701, 763
{$I-} 546, 575, 688
{$M} 546, 688
{$N+} 329, 332, 335
{$O+} 763
6845 CRT controller 80
80287 526
80387 526
8087 526, 566
8x8 font 30, 49, 72

A

Absolute screen coordinates 22
Action codes 493
Active display page 260
Active page 263, 282
Active window 84
Alarm 715
Alternate stack 525, 529, 531, 540, 544
ASCIIZ 421, 465
Assembly language routines 3
Assumptions 6
Attributes, file 502
Attributes, video 33, 46, 55, 84, 721, 722
Auto-detection of display adapters 29
AutoAdvance 358
AutoNumLock 359

B

Background character 262, 282
Background task hook 753
Background tasks 311, 745, 757
BCD 325, 665, 710
Benchmark 630
Bickel, Michael 431
Binary Coded Decimal 665, 710
Binary Editor 308
BINOBJ.EXE 177
BiosScroll 70, 71
Blank trimming 373
BLDGUIDE.EXE 5
Blinking text 73, 722
BONUS 5
Boyer-Moore algorithm 426, 427
BSORT 735
BufLen 21
Burns, Philip R. 666

C

Calculated Fields 306
CGA emulation 78
ClearFirstChar 360
Color mapping 29, 84, 260
Command entry pointer 534, 549, 562
Command, last 304
Compaq 21
Compiler support 9
Compression 60
CompuServe 18
COMSPEC 487
Conditional compilation directives 6
CONFIG.SYS 497
Context 587, 600, 601
Context-sensitive help 168, 311, 745, 753, 760
Control characters 154, 302, 361
Conversion routines 288, 313
Copying B-Tree Filer for C 17
Critical error 527, 546, 551, 561, 575
CRT, compatibility of TPCRT with 22
CRT unit 19, 262
Currency 298, 320, 354
Current time 385

Current window 84
Cursor 51, 58, 67, 72, 84, 100
Cursor, hidden 54
Cursor mask 201
Cursor position 80
Cursor scan lines 32

D

Data entry flow 287
Data entry screens 745
Data entry screens, nested 307, 334
Data entry window 308, 362
Data types 285
Database Toolbox 390, 399, 579
Date arithmetic 378, 381
Davis, Dan T. 234
Day of the week 382, 390
DBase 295
Delimiters 362
DesqView 19, 22, 29
Dettmann, Terry 7
Devices 505
DIFF 741
Directories 510
Directory, change 256
Directory name 424
Directory window 252
DirectVideo 24
Dirty pages 657
Disabling a TSR 558
Disabling menu items 122
Disk class 498
Disk sectors 507, 517
Disk Transfer Address 510, 551
Display adapters 27, 30
Display pages 19, 21, 30, 75, 215, 229, 260, 265, 282
DOS busy flag 559
DOS critical flag 559
DOS services 542, 545, 550, 555
DOS shell 493, 571
DOS unit 487
Double DOS 29
DTA 510, 551
Duncan, Ray 7

E

Edit buffer 402
Edit window 402
Editing commands 153, 155, 165, 300
Editor control block 402
Editor Toolbox 308
EGA 30, 36, 47, 49, 72, 74, 75
EKFIX.COM 520
EMM 587
EMS 548, 587, 629, 630, 765
EMS page frame 594, 599
EMS pages 587, 591, 639
Emulator 688
End Of Interrupt 539
Enhanced keyboards 488, 518
Enumerated types 307
Environment variables 501
EOI 539
Error Handling 310
Event handler 205, 213, 228, 243
Exception interrupts 568, 570, 571
ExecDos 67, 279, 487, 571
Exit commands 118, 158, 240, 289, 294, 300
Exit procedures 6, 280, 592, 688, 690
ExitProc 6
Expanded memory 587, 741
Expanded Memory Manager 587
Expanded Memory Specification 587
Exploding window 86
Extended memory 587, 606, 741
Extension 435, 439, 445

F

FAR model 6, 493, 586
FASTUPD.nnn 4

Field attribute, change 341
Field delimiters 362
Field editors 300
Field ID 293
Field movement commands 300
File pointer 515
FileMode 495
Files supplied 4
Filter 504
Flicker 22, 28, 57, 70, 71
Floating point emulation 527, 548, 566
Floating point overflow 666
Formatted strings 422, 440, 484
Formatting characters 294, 379
Frame characters 50, 74, 100, 362
Free list 488

G

GRAPH unit 23
GRAPH3 22
Graphics 21, 24, 30, 63, 67
GUIDE.DOC 5

H

Handles, EMS 591
Handles, file 489, 497
Handles, ISR and popup 525, 528, 535
Handles, screen 260
Hard disk 3
Heap overflow 688
Heap size 546
HeapError 688
Help file directives 170
Hercules 27, 30
Hercules graphics card 28
Hercules graphics mode 52, 75
Hercules InColor card 78
Hexadecimal arithmetic 543
Hidden field 315
Hotkeys 115, 543, 554, 708, 726
Huge pointers 588, 607

I

Implementation 6
Inline macros 6, 609, 687, 692
INSTALL 4, 11
Installable keyboard hook 753
Installation program 756
Interface 6
Interrupt procedures 6, 525, 528, 693, 698
Interrupt Service Routine 525, 528, 535
Interrupt vectors 6, 535
Interruptible 759
ISR 525, 528, 535
ISR handles reserved 528

J

Jourdain, Robert 7
Julian date 380
Jump table 695, 701

K

Key Assignments 314
Key Codes 518
Keyboard buffer 77
Keyboard handling 753
Keyboard macros 609, 610
Keyboard status flags 54

L

LBR file format 35
Letter set 431
LHARC.EXE 4
Libraries 35
Line editor 153
Linear addresses 588, 602, 607
Links, field 356
Literal characters 294

Logical pages 594, 599
Lotus-style menus 117

M

Macro Assembler 3, 667
Macro editor 627
Macro engine 617
Macro file 621, 706
Macro name 614
Macro recording 619
Macros, keyboard 609, 610
Macros, nested 609
MAKE 4, 11
Make file 10
MAKEHELP 169, 175, 762
MAKEMENU 115, 136, 142, 201, 724
Map colors 29, 84, 260
Mapping context 587, 600, 601
MASM 3, 667
MCGA 30, 75
Memo fields 402
Memory management 581
Memory requirements 3
Memory resident programs 6
Menu ID 144
Menu item 115
Menu item, disabling 122
Menu library 142, 144, 152, 727, 733
Menu selection keys 150
Mickey 209, 214
Mickey-to-pixel ratio 209, 227
Microsoft Windows 29
Mistakes, common 581, 631
Modules 548, 560
Monitor 3
Mouse buttons 202, 221, 225
Mouse cursor 200, 216, 230
Mouse drivers 208, 723
Mouse droppings 204
Mouse event handler 205, 213, 228, 243
Mouse support 115, 126, 154, 184, 195, 200, 231, 309, 311, 351, 720, 743
Mouse window 203, 214, 224
Multiple-choice editor 300
Multiple-choice fields 306, 328
Multitasking 29

N

NEAR 6
Network 495
Normalized pointer 702
Norton, Peter 6
Numeric coprocessor 526, 688
Numeric editor 300, 367
Numeric keypad 64

O

OmniView 19, 22, 29
On-screen clock 716
OPSTACK 548
OS/2 551
Overflow errors 568
Overlay errors 688
Overlay Manager 763
Overlays 548, 751, 763

P

Page frame 594, 599
Password mode 368
PATH 442, 487, 496
PCALC 710
Periscope 546
Phillips, Carley 22
Physical pages 594, 599
PIC 539
Pick list orientation 234, 247
Pick lists 231, 745
PickHorizontal 234
PickSnaking 234

PickVertical 234
Picture mask 254, 294, 378
Playback delay 708
Pop ticker 564
Popup dispatcher 542
Popup handles 543
Popup routines 542
Popup window 50, 749
Post-edit routine 289
Pre-edit routine 288
PREF 201, 721
PRINT.COM 487, 506, 513, 709
PRINTDOC.EXE 4
Printer errors 577
Program Segment Prefix 551
Programmable Interrupt Controller 539
Prompt attribute, change 341
Protected fields 235, 306, 342, 371
Protected mode 9, 60, 587
Protection 343
PS/2 3
Pseudo-scan codes 203, 221, 225, 721, 756
PSP 551
PTIME 715
PTRPT 718
Purchase Agreement 17

Q

Quicksort 579

R

RAMDRIVE.SYS 588, 589
Random access 487
Read-only mode 322
READ.ME (DEMO) 4, 13
READ.ME! (BONUS) 5
Redefinable keyboard 231
Redirection 738
Reentrancy 206, 408, 529, 550
Registers 544
ReinitCrt 279
ReleaseEms 765
REP 738
Required field 305, 372
Required status 343
Restore a vector 538
Right justification 372
Run-length encoding 60
Runtime errors 546, 568, 687, 688, 690

S

Scan code 64, 543
Scan lines 37, 74
Screen coordinates 46
Screen mask 201
Scroll bar 181, 195, 231, 239, 309, 722
Scrollable entry screens 309
Scrolling 28, 54, 70, 71, 308
Segmented addresses 588, 607
Select menu items 140
Semi-literal characters 294, 295, 379
Shadowed window 87
Shift keys 543
SideKick 520, 550
SideKickPlus 550
Slider 239
SMACS 706, 715
Snow 22, 28, 57, 70, 71
Sortable string 390, 396, 399
Sorting 579, 694, 735
Soundex 421, 431, 455
Stack and heap usage 547
Stack overflow 688
Stacked windows 83, 84
Stacks 525, 529, 531, 540, 544
Standard input 489, 505
Standard interface 548, 560
Standard output 489, 505
Status word 568, 570
Strenczewilk, Don 427
String attribute, change 344

String comparison 433
String editor 300
String manipulation 421
Subfield 299
Submenu 115
Support, telephone or CompuServe 18
System requirements 3

T

TaskView 29
TASM 3
TDebugPLUS 546
Telephone support 18
Terminate and stay resident 6, 525, 542, 565
Text editor 402
Text file device driver 484
Text files 487
TFIND 736
Tiled windows 84
TopView 29
TPC.CFG file 11
TPKEYS 756
TPRO.MAK 4, 11
TPTIMER 516
TPU files 9
TSR 6, 21, 67, 207, 279, 493, 525, 542, 565, 587, 758
Turbo Assembler 3, 667
Turbo Debugger 546
Turbo Extender 763
TURBO3 22
Turbo3StyleColors 262
TURBO5.MAC 706

U

Unhidden 54
Unit dependencies 751
Unit initialization blocks 6
Unit name override 629
Units 6
Unloading a TSR 549, 558
User-defined display function 141, 152
User-defined types 312, 339
USES statement 6

V

Validation routine 313, 344
VDISK.SYS 588
VERIFY 512
VGA 30, 47, 49, 72, 75
Video attributes 33, 84
Video memory 31, 57
Video mode 24, 30, 50, 67
Video pages 19, 21, 30, 75, 215, 229, 260, 265, 282
VideoSegment 21
Virtual screens 20, 89, 260, 264, 266, 749
Visible display page 260, 282

W

Warranty period 18
White space 461, 474
Wilton, Richard 7
Window coordinates 28, 281, 284
Window, popup and tiled 83, 100
Window stack 83
Windows, Microsoft 29
WINEXMPL 749
WINWOW 749
Word delimiters 463
Word wrap 402, 464
WordStar 153, 165
Wrap mode 301, 375

Z

Zenith 21
Zero-based arrays 630

NOTES

NOTES

NOTES

NOTES

NOTES

NOTES

NOTES

NOTES

NOTES

NOTES

TURBO PASCAL

A Powerful Library for Turbo Pascal

Turbo Professional is a library of over 600 essential procedures and functions, optimized specifically for non-object-oriented Turbo Pascal programming. You get over 50,000 lines of perfected code including:

- Pop-up resident routine
- Windows and menus
- Data entry routines
- BCD arithmetic
- Interrupt service routines
- Context sensitive help
- EMS and extended memory access
- Long strings
- Large arrays
- Macros
- Fast sorting
- Much, much more

Powerful Programs, Faster and Easier

With these routines, you'll develop more powerful programs, and save time and effort. You can concentrate on the creative aspects of the job and let Turbo Professional handle the low level chores.

Hot Demo Programs, Too

Turbo Professional also includes more than ten commercial quality programs that illustrate how to use the routines in your programs. You'll find a wealth of specific examples in:

- A memory-resident keyboard macro processor
- A memory-resident programmer's quick reference chart
- A memory-resident programmer's calculator
- A memory-resident programmer's time and billing program
- An interactive menu maker that writes source code for you
- A bullet proof full-screen data entry demo with pop up help

Documentation: Clear and Complete

Turbo Professional includes clear, comprehensive documentation and pop-up help. The calling syntax, dependencies, and operation of every routine are described. You get free expert telephone and CompuServe support. Full source code is included and you pay no royalties when you use these routines.

Turbo Professional requires Turbo Pascal 4.0, 5.0, 5.5, 6.0, 7.0 or Borland Pascal 7.0. MS-DOS 2.0 or later and an IBM, PC, XT, AT, PS/2 or compatible are required.

TURBO Power

TurboPower Software
P.O. Box 49009
Colorado Springs, CO 80949-9009
CompuServe I.D. 76004, 2611

© TurboPowerSoftware 1993